HOS

Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 531
Cambridge, Massachusetts 02140
617 661 8900

CONTACT:
Ralph Specht
Higher Order Software, Inc.
(617) 661-8900

HIGHER ORDER SOFTWARE RELEASES USE.IT®

ENHANCEMENTS FOR THE IBM PRODUCT LINE

CAMBRIDGE, MASS., March 3, 1986 -- Higher Order Software, Inc. (HOS) today announced shipment of Release 2.0 of their USE.IT software product for the IBM MVS environment. USE.IT automates the requirements and design process which results in substantial savings in software development and maintenance. Automating the process also eliminates errors which are so prevalent in a manual design and implementation process.

With USE.IT, system developers create specifications on a graphics terminal with a technology paralleling computer-aided design and manufacturing tools (CAD/CAM). Specifications are automatically analyzed to detect logical errors. When all logical errors are corrected, USE.IT automatically constructs source code and documentation. "With traditional design methods, the end-user describes what he or she wants the software to do. But the requirements, specifications,

(more)

design, and code are written in different languages, and with different procedures. Each language translation introduces new errors into the development process. By the time the software is finished, it may not bear much relationship to what was originally requested. With USE.IT, one language is used to define requirements and specifications and the source code is constructed automatically from the design." said a company spokesman.

Specific enhancements in this release include: the ability to integrate data dictionary definitions with USE.IT designs and implementations; increased use of local terminal capabilities to reduce machine resource requirements and further improve user productivity; and the capability, in multi-site design/development environments, to capture designs and implementations and transfer them from one location to another.

In addition to the IBM MVS environment, the USE.IT product line is also available for VAX/VMS users.

For further information contact: Higher Order Software, Inc., 2067 Massachusetts Avenue, Cambridge, MA 02140. Tel: (617) 661-8900.

# # #

Higher-Order Sw
or - HOS

new f.

Touchstone Engineering Corporation

GamePlan - A New Approach To Management Planning

35 Medford Street
Post Office Box103
Somerville, MA 02143
(617) 628-3200

What is GamePlan?

GamePlan is a planning environment for professionals who plan and manage major projects and programs.

You state your top-level assumptions and objectives. Then, as you devise a strategy to achieve those objectives, GamePlan provides you with an optimal tactical plan.

The "intelligence" built into GamePlan helps you to avoid tedious planning activities. You concentrate on "what" you want to accomplish. GamePlan computes the "how".

As you identify key management roles and responsibilities, GamePlan helps you set up guidelines for your management team. Each team member can then supply details about specific parts of your plan. GamePlan helps you to integrate these sub-plans.

As your plan unfolds, Gameplan helps you to maintain a systematic approach. GamePlan will tell you if your sub-plans are consistent with your higher level objectives.

As you know, the only thing constant in planning is change. Because GamePlan is so highly integrated, you can, for the first time, do real "what if" analysis.

The technology underlying GamePlan is a powerful new approach to activity and resource integration, coupled with frame-based techniques derived from AI research. Unlike the conventional network-based project management technology, GamePlan uses integrated hierarchies. The introduction of integrated hierarchies allows for a new dimension of computational intelligence and flexibility to the planning professional.

Better planning today -- with GamePlan.

Product Overview

GamePlan has been designed to meet the following objectives:

- You should be able to plan as you manage - and manage as you plan.
- You should be able to concentrate on creativity - and leave the tedium to the computer.
- You should be able to have qualitative and guantitative summaries at your fingertips.
- You should be able to evaluate alternative plans guickly and interactively.

To meet the first objective, we designed a user-interface that closely mirrors the real world of management. GamePlan is composed of three integrated hierarchies.

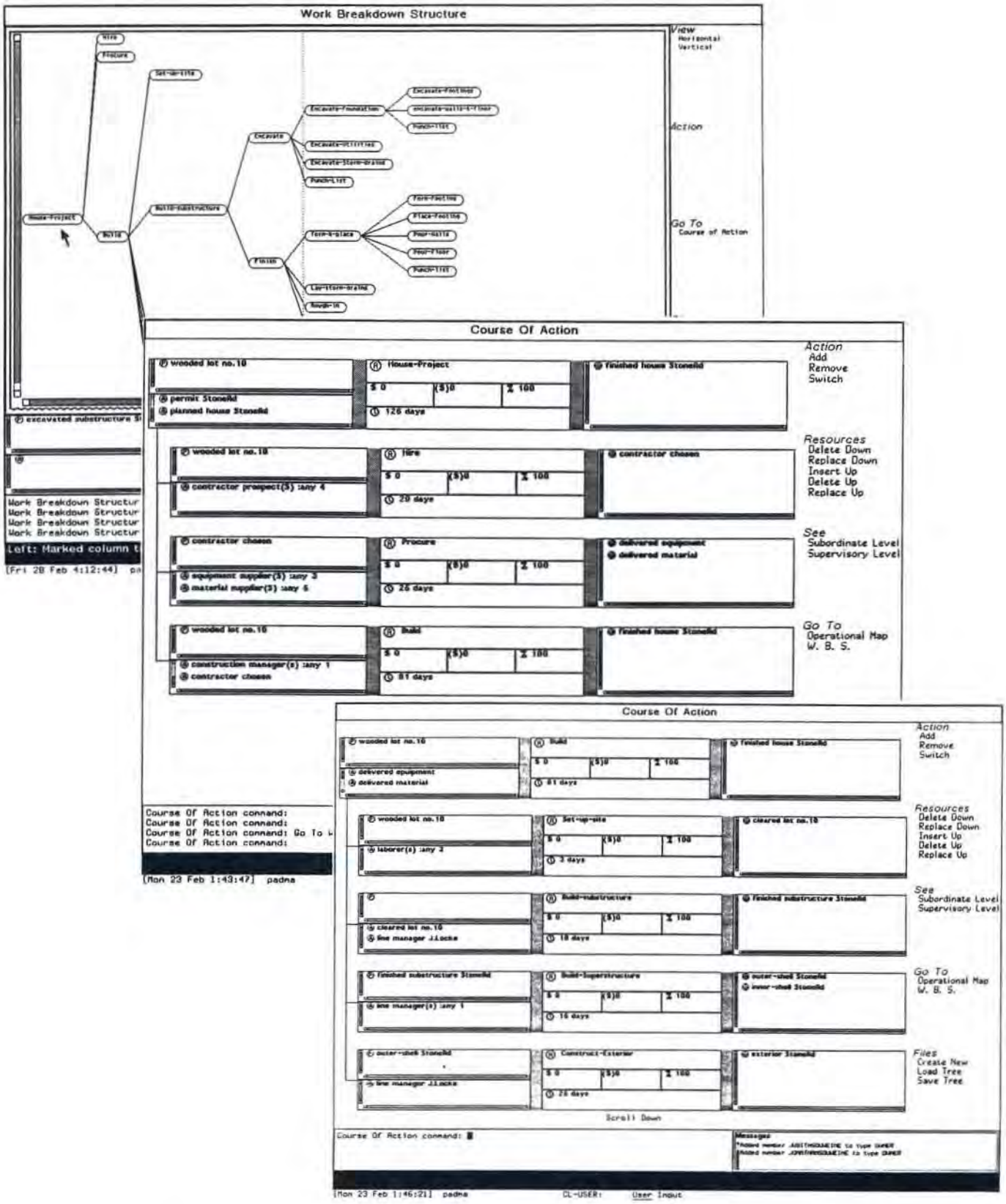
To meet the second objective, we have provided logical interface analysis for each of the three hierarchies. You may design your plan top-down, bottom-up or middle-out. GamePlan assures you an internally consistent plan.

To meet the third objective, you can consolidate or expand information at any planning level and from any planning view.

To meet the fourth objective, you can play with confidence factors, manipulate resource limits and reconfigure actions dynamically.

Any plan designed with GamePlan is composed of a course-of action, an operational-map and a supporting set of resource definitions.

> The *course-of-action* is a hierarchy of planning actions and corresponding assumptions and objectives.
>
> The *operational-map* is a hierarchy of actions and corresponding resources distributed over a calendar period.
>
> You define the types of resources you want as a hierarchy of types and sub-types within the *Resource Definition Facility* .



THE COURSE OF ACTION: A hierarchy of planning actions.

Course of Action

On the course-of-action, you divide your strategy into discrete actions. Some or all of these actions may have their own subordinate actions. Subordinate actions, in turn, may have subordinate actions of their own.

Through the course-of-action, you set priorities and assign assumptions and objectives. Each action may have its own assumptions and objectives. Assumptions describe how many or which resources an action needs: objectives describe the resources an action produces.

If an action has no subordinates, you assign the time needed to meet the objectives of that action. If an action has subordinates, GamePlan computes the time for you.

The course-of-action meets GamePlan's design objectives as follows:

OBJECTIVE: **A user-interface that closely mirrors the real world of management.**

The course-of-action is integrated. This assists you to manage by objectives.

Look at the course-of-action example on the opposite page. Notice that the objectives of the supervisory action appear as objectives of one of the subordinates. The provided assumptions of the supervisory action are used by the subordinates. The subordinates work together to meet the objectives of the supervisory actions.

The course-of-action is modular. This makes it easy to reconfigure your plans.

Notice that each action, be it a supervisor or subordinate follows the same format. Each is self-contained. Each can be viewed as an individual "contract" with its own assumptions and objectives.

OBJECTIVE: **Automatic logical analysis to eliminate tedium.**

GamePlan's built-in connection rules are used to check both the interface logic and constraint logic. If there is an error, GamePlan points you directly to the problem.

The interfaces among supervisory and subordinate actions are automatically analyzed by GamePlan. This makes it easy to add or delete actions.

All the interfaces among assumptions and objectives for any one action are analyzed by GamePlan. This makes it easy to add resource constraints to any action.

OBJECTIVE: **Consolidate or expand information as needed.**

The Work Breakdown Structure provides you with a summary of the course-of-action by action name. From this view, you may select any course-of-action. You may also use this view to enter or change any action-frame's information.

At each action-frame, you get a summary of assumptions, objectives, duration time, confidence factor, cost and revenue for that action.

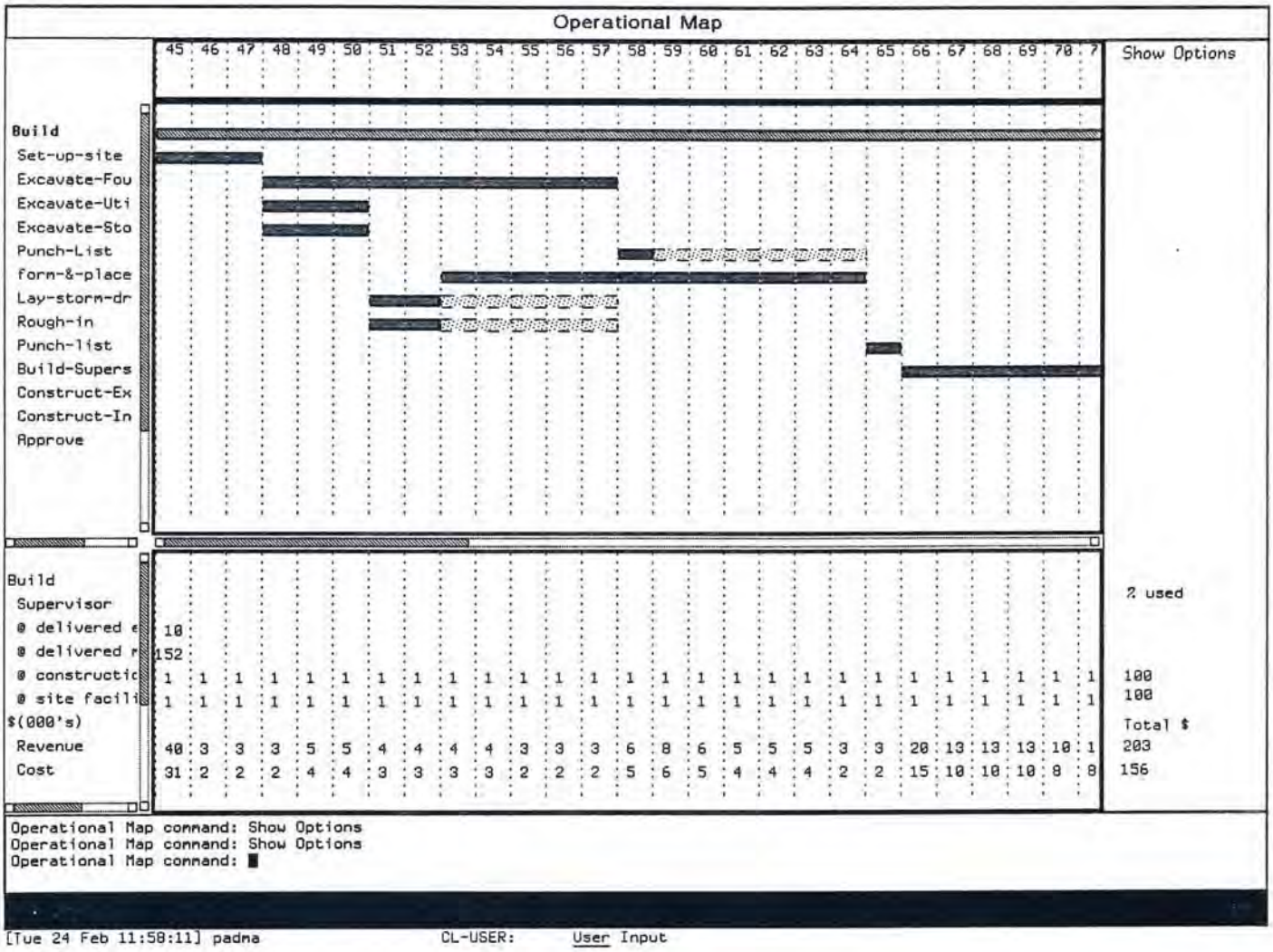
For any course-of-action, you may expand the view of the action-frame to include its set of subordinate actions. You may use the view to enter or change any action-frame's information.

OBJECTIVE: **Real "What If" Analysis**

At any action-frame, you may assign a confidence factor, expressed as a percentage. This factor is your estimate of how confident you are that you can achieve your objectives as planned.

Whenever you wish, you may request GamePlan to show you the best or worst case analysis of your plan. To compute the best case scenario, GamePlan assumes you have 100% confidence in each action's estimate. To compute the worst case scenario, GamePlan "rolls-down" cumulative confidence factors from the top of your course-of-action to the lowest subordinate actions.

You may, of course, change confidence factors whenever and wherever you wish and request GamePlan to compute various scenarios.



THE OPERATIONAL-MAP: Action and Resource Distribution for your course of action over a calendar period.

Operational Map

When you have defined a course-of-action, GamePlan can find both the optimal schedule and the optimal resource allocations for all of the actions. The operational-map shows you the schedule and resource allocations over time.

GamePlan infers the precedence relationships among actions by analyzing your assumptions and objectives. When you describe the type of resource you want and how many you need, GamePlan allocates the resources you request and schedules the action in accordance with your priorities and the available resources.

GamePlan computes how resources can best be shared. Shared resources may include time. The overhead resources of a supervisory action are shared by all its subordinates.

The operational-map meets GamePlan's design objectives as follows:

OBJECTIVE: **A user-interface that closely mirrors the real world of management.**

The operational-map is integrated. This assists you in effective utilization of upper, middle and lower management resources.

Look at the operational-map example on the opposite page. Notice that the time and resource distribution are shown for both supervisory and subordinate actions. Whenever you adjust a subordinate action, you may affect the length of time over which you need supervisory activity. You may adjust supervisory actions without bothering with the details.

The operational-map is modular. This makes it easy to adapt actions to new project environments.

Whenever you use one course-of-action as part of a new project, it can be viewed as a reusable "template" for dynamic resource allocation.

OBJECTIVE: **Automatic logical analysis to eliminate tedium.**

The timing interfaces among supervisory and subordinate actions are analyzed automatically by GamePlan. This eliminates the need for you to define precedence rules.

The resource interfaces are analyzed automatically by GamePlan from your assumptions and objectives. This eliminates the need for you to manually allocate resources.

OBJECTIVE: **Consolidate or expand information as needed.**

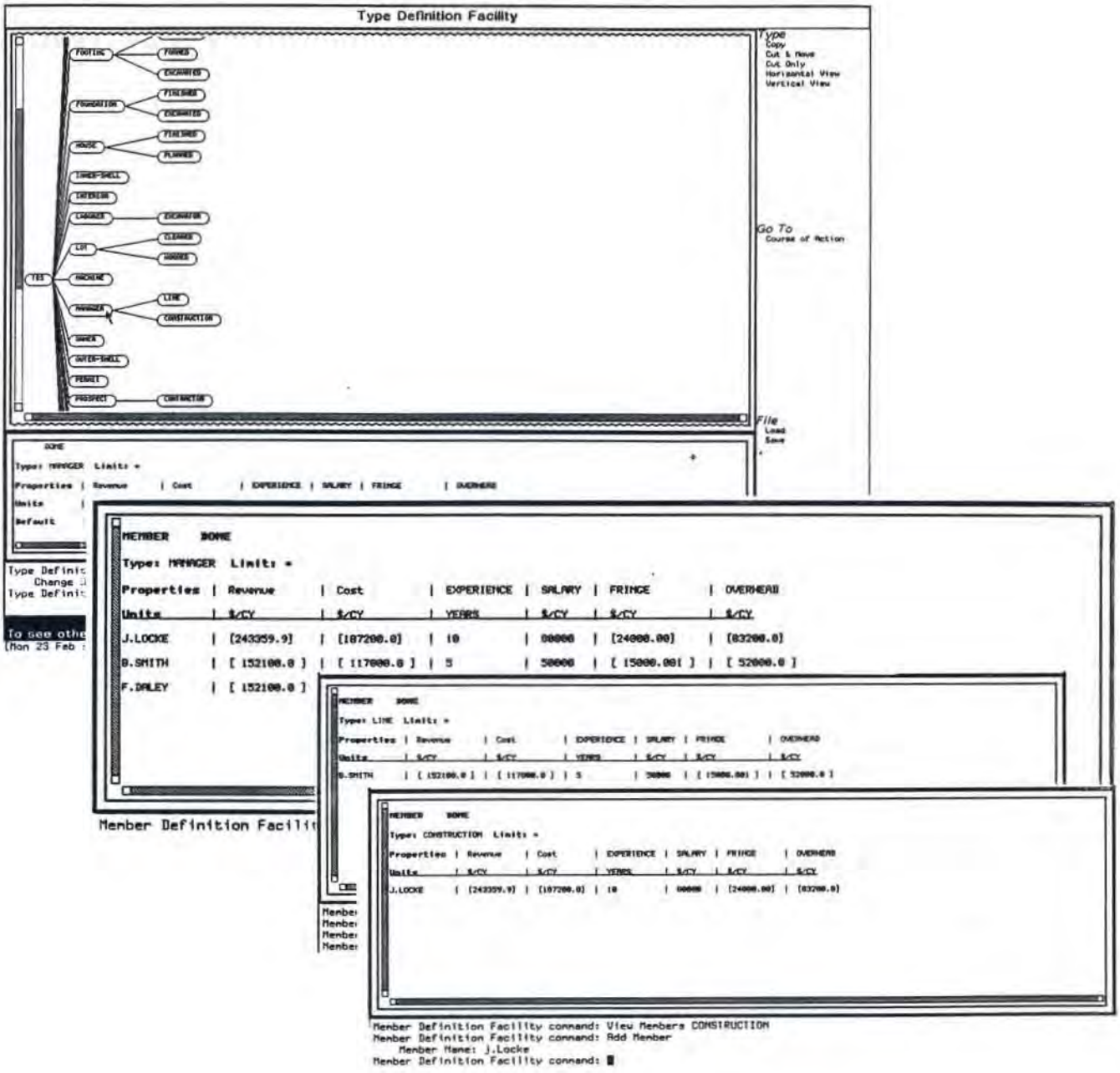
The Action Schedule may be viewed for one planning level or expanded in depth to the lowest subordinate actions.

The resource "spreadsheet" shows you resource utilization and associated revenues, costs and profit. You may request a summary for a particular action or an expansion separating supervisory from subordinate resources. For each summary resource, you may expand by sub-type until you see each member.

OBJECTIVE: **Real "What If" Analysis**

At any scheduled action, the operational-map shows you the "slack" time. You may interactively adjust that action over time within the slack period. Whenever you do so, you will immediately see the effects on resource utilization and associated dollars.

You can then fine-tune your resource levelling interactively.



RESOURCE DEFINITION FACILITY: A means to define resource objects.

Resource Definition Facility

With the Resource Definition Facility, you may organize your resources in types that have common attributes. Some or all of these types may have sub-types.
Sub-types may have sub-types of their own.

Types inherit the attributes of their super-types. They may have additional attributes of their own.

Members of a type are also members of their super-type. If a type has sub-types, the members of one sub-type may be members of another sub-type.

The Resource Definition Facility meets GamePlan's design objectives as follows:

OBJECTIVE: **A user-interface that closely mirrors the real world of management.**

The Resource Definition Facility is integrated. This assists you to effectively utilize the same resource as it plays different roles.

Look at the Resource Definition Facility example on the opposite page. Notice that construction manager is a sub-type of manager and line manager is a sub-type of manager. Joe could be a construction manager. He could also play the role of line manager. Joe inherits a manager's attributes. In addition, he may have construction manager attributes and line manager attributes. On the course-of-action, you may use Joe as a construction manager for one action and as a line manager for another action.

The Resource Definition Facility is modular. This assists you to effectively utilize different resources within the same type.

Each type is self-contained with its own attributes and members. You may add or remove members whenever you want to. You may also disconnect a type from a super-type and connect it to another type. Whenever you do so, the type inherits the new super-type's attributes; but it keeps its self-contained attributes and members.

OBJECTIVE: **Automatic logical analysis to eliminate tedium.**

The interfaces among types are automatically analyzed by GamePlan. This makes it easy for you to add or remove types.

All the interfaces among the attributes for any one type are analyzed by GamePlan. This makes it easy for you to add or remove attributes.

As GamePlan checks the interface logic, it will explain to you exactly where an error occurs.

OBJECTIVE: **Consolidate or expand information as needed.**

The Type Breakdown Structure view of the Resource Definition Facility gives you a summary by type-name. From this view, you may select any type. You may also use this view to enter or change any type's attributes.

For each type, you get a summary of the limit, properties and defaults.

For each type, you can view a table of property values by member. You may use this view to add members, delete members or change property values for any member.

OBJECTIVE: **Real "What If" Analysis**

For any resource type, you may assign a limit, expressed as a whole-number. GamePlan will add members to a type for you whenever the members you have specifically identified are insufficient to execute your plan in the shortest possible time. Whenever you assign a limit, you restrict the number of resources available for GamePlan to use in allocating resources for your course-of-action.

The default assumed by GamePlan is that an unlimited number of resources can be added. If all types are defaulted, you will see how many additional resources you will need to execute your plan in the shortest possible time.

By changing type limits, you can evaluate resource utilization vs time to complete.

GamePlan's Working Environment

As a planning professional, you need to be able to use computing power intuitively. GamePlan, available on any of the Symbolic 3600 family of workstations supports that need.

You use high resolution graphics to communicate rapidly and interactively. You can use a mouse or a command language, as you prefer.

You use a large screen which gives you more information at a glance.

You take advantage of sufficient processing power to give you the answers you need quickly.

GamePlan combines state-of-the-art software technology with state-of-the-art hardware technology to provide you with the ultimate planning machine.


HOS

OS
Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 531
Cambridge, Massachusetts 02140
617 661 8900

MARGARET H. HAMILTON

PRESIDENT AND CHIEF EXECUTIVE OFFICER

Margaret H. Hamilton co-founded Higher Order Software, Inc. (HOS) in 1976 and has served as its president and chief executive officer since that time.

She is co-creator of a formal axiomatic system definition and development methodology called Higher Order Software, and of the functional life cycle model of this methodology.

Prior to founding HOS, she directed the Apollo on-board flight software effort for both the command module and the lunar module systems while division leader

HOS
Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 531
Cambridge, Massachusetts 02140
617 661 8900

CONTACT:
Karen Kosko
Higher Order Software, Inc.
(617) 661-8900
-or-
Paula Neely
Newsome & Company, Inc.
(617) 426-4300

FOR BUSINESS EDITORS

HIGHER ORDER SOFTWARE INTRODUCES COMPUTER-AIDED SOFTWARE DEVELOPMENT TOOL FOR IBM ENVIRONMENT

CAMBRIDGE, Mass., April 15, 1985 -- Higher Order Software, Inc. (HOS) today announced that the automated software development tool, USE.IT, now runs on IBM mainframe computers. USE.IT makes the entire process of developing and maintaining software systems shorter and more logical, reduces errors, and simplifies the design process. "The productivity of the data processing department increases, and costs are cut with USE.IT," said Margaret Hamilton, president and chief executive officer

Headquarters
2067 Massachusetts Avenue
Post Office Box 531
Cambridge, Massachusetts 02140
617-661-8900
Telex 951253 HOS INC CAM

Eastern Region
551 Fifth Avenue
Suite 1110
New York, New York 10176-0027
212-490-8721

Western Region
8445 Freeport Parkway
Suite 420
Interfirst Place
Irving, Texas 75063
214-257-3758

HOS
Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 531
Cambridge, Massachusetts 02140
617 661 8900

MARGARET H. HAMILTON

PRESIDENT AND CHIEF EXECUTIVE OFFICER

Margaret H. Hamilton co-founded Higher Order Software, Inc. (HOS) in 1976 and has served as its president and chief executive officer since that time.

She is co-creator of a formal axiomatic system definition and development methodology called Higher Order Software, and of the functional life cycle model of this methodology.

Prior to founding HOS, she directed the Apollo on-board flight software effort for both the command module and the lunar module systems while division leader at the Charles Stark Draper Laboratory, Inc. At Draper, she was directly responsible for several basic research and advanced tool development and application efforts, including the SKYLAB flight software and the integration of requirements for the space shuttle flight software.

Hamilton has concentrated on the relationships between basic research and system applications. She has conducted research in areas such as techniques for defining reliable

-more-

systems; operating system design; man-machine interface design; error detection and recovery design; and methods for managing, integrating, developing, and maintaining large-scale, real-time, multiprogrammed, multiprocessed system development activities.

Hamilton holds a bachelor of arts degree in mathematics from Earlham College.

# # #

0491d/032785/GF

HOS
Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 830
Cambridge, Massachusetts 02140
617 661 8900

COMPANY FACT SHEET

Company:	Higher Order Software, Inc. 2067 Massachusetts Avenue Cambridge, MA 02140 (617) 661-8900
Founded:	1976
Founders:	Margaret H. Hamilton, Saydean Zeldin
Product:	USE.IT, a computer-aided software design and production tool, analogous to computer-aided design and manufacturing (CAD/CAM) tools, for the IBM mainframe and DEC VAX minicomputer markets. USE.IT lets software designers create specifications that are logically consistent and complete before a single line of code is produced, then automatically generates running programs. Additional aspects of HOS' business include education and consulting.
Sales:	HOS has sales offices in New York City, Dallas, and Atlanta. USE.IT also is distributed by Metra-HOS, Brussels, which is a joint venture between Sema-Metra, Paris, and HOS.
Officers:	Margaret H. Hamilton, president and chief executive officer; Saydean Zeldin, executive vice president and chief financial officer; Thomas D. Lutz, vice president of marketing and sales; Tom L. Kev, vice president of development; David Blohm, vice president of finance and administration
Ownership:	Privately held; venture capital financing of $9.2 million
Investors:	Alex. Brown & Sons, Inc.; Cazenove & Co.; Emerging Growth Partners; Frontenac Venture Co.; Greylock Management Corp.; Henry & Co.; Henry L. Hillman; James Martin; Merrill Lynch Venture Capital Ltd.; Newcastle Co. Ltd.; Sears Investment Management Co.; J.F. Shea Co., Inc.; Venrock, Inc.
Employees:	75

4122c/040385/GF

HOS
Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 541
Cambridge, Massachusetts 02140
617 661 8810

THOMAS D. LUTZ

VICE PRESIDENT OF MARKETING AND SALES

HIGHER ORDER SOFTWARE, INC.

Thomas D. Lutz has been vice president of Higher Order Software, Inc. since October, 1984.

Prior to joining HOS, Lutz was principal and director of education and communications at Nolan, Norton and Company, a management consulting firm that specializes in information technology in business. He was responsible for the marketing and production of all educational products and services, including public education courses and seminars, in-house education programs, video-based education and education consulting.

Previously, Lutz was director of education for ITT Programming. At ITT, Lutz spearheaded an entrepreneurial, business approach to providing educational services for ITT personnel and customers in the information technology industry.

Lutz also spent seven years as the head of information systems for the Mayo Foundation, where he was responsible for developing and marketing clinical, research and administrative systems.

-more-

Lutz began his career at IBM as an applied scientist and manager at the Systems Research Institute. During his 14 years with IBM, he founded and directed the IBM Systems Science Institute.

Throughout his career, Lutz has served as adjunct faculty member to several graduate schools, including the University of Minnesota, the Pratt Institute and the University of Newcastle (U.K.). He is an internationally recognized lecturer and is the author of several DELTAK video journals on systems management.

Lutz holds a bachelor of science degree in mathematics from South Dakota School of Mines and Technology and an masters of science degree in operations research from New York University.

# # #

0384d/040385/NEELY

HOS
Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 531
Cambridge, Massachusetts 02140
617 661 6900

CONTACT:
Karen Kosko
Higher Order Software, Inc.
(617) 661-8900
-or-
Paula Neely
Newsome & Company, Inc.
(617) 426-4300

FOR BUSINESS EDITORS

HIGHER ORDER SOFTWARE INTRODUCES COMPUTER-AIDED SOFTWARE DEVELOPMENT TOOL FOR IBM ENVIRONMENT

CAMBRIDGE, Mass., April 15, 1985 -- Higher Order Software, Inc. (HOS) today announced that the automated software development tool, USE.IT, now runs on IBM mainframe computers. USE.IT makes the entire process of developing and maintaining software systems shorter and more logical, reduces errors, and simplifies the design process. "The productivity of the data processing department increases, and costs are cut with USE.IT," said Margaret Hamilton, president and chief executive officer of HOS.

With USE.IT, system developers create specifications with a Graphics Editor in an easy-to-understand graphical format. Specifications are automatically analyzed to detect logical errors. When all logical errors are corrected, USE.IT automatically generates source code, bypassing error-prone manual programming processes.

-more-

"Computers have been used for years to automate procedures and increase productivity in other areas, but the development of software itself has not been automated," said Hamilton.

It is 10 to 100 times more expensive to correct an error in the testing stage than to find and correct it as the software is being designed. With so much of their resources dedicated to maintenance, data processing departments are falling further behind in developing new applications.

"It is becoming clear that the ability of a company to compete depends largely on its ability to process information more efficiently than its competitors," said Hamilton. "Companies are demanding complex new applications, such as manufacturing or financial control systems, to fulfill their strategic requirements. Yet transforming these requirements into reliable software is time-consuming and expensive at best, and in some cases impossible, using current development tools."

USE.IT approaches this problem by applying computer-aided design to the software development process. This is analogous to computer-aided design and manufacturing tools (CAD/CAM). Just as CAD tools let the designer model a product and correct design errors before production begins, USE.IT lets software designers create specifications that

are logically consistent and complete before a single line of code is produced. And just as CAM tools automate production, USE.IT automatically generates running programs based on the specifications.

"USE.IT also addresses the problem of software that does not meet the need for which it was ostensibly designed," said Hamilton. "With traditional design methods, the end-user describes what he or she wants the software to do. But the requirements, specifications, design, and code are written in different languages, and with different procedures, using manual processes. By the time the software is finished it may not bear much relationship to what was originally requested."

With USE.IT, the end-user and the software developer can work together at a terminal to develop specifications in a simple, easy-to-understand graphical format, using the end-user's terminology. Specifications are automatically analyzed for logical correctness. At any stage in the specification process it is easy to create a prototype to show the end user how the software will work.

"The ability to create and run prototypes quickly enables the end user to participate closely with the software developer in the specification process and eliminates the problem of systems that do not do what the user wanted them to do," Hamilton said.

When applications are completed, corrected, and tested, USE.IT automatically produces computer code,

bypassing traditional manual programming. If a completed application must be changed for any reason, the changes are made to the specification and code is regenerated.

International Data Corp. (IDC) estimates that large-scale systems users spent approximately $126 million on software design and development tools such as program and application generators and database management systems in 1984 and predicts sales of $500 million by 1988. "These figures do not even take into account complete automated development tools such as USE.IT," said Hamilton. "Obviously, there is a large and rapidly growing market for products that can ease the growing development burden of data processing departments."

After two years of successful experience on DEC VAX systems, USE.IT is available for IBM mainframes running the MVS operating system and presently supports the COBOL language. Interfaces exist for IMS and CICS. Releases planned for late 1985 include support for the FORTRAN language and IDMS data base management system. The introductory price of USE.IT is $95,000 and it will be available in April, 1985.

For further information contact: Higher Order Software, Inc., 2067 Massachusetts Avenue, Cambridge, MA 02140. Tel: (617) 661-8900.

# # #

4391c/032085/GF

HOS
Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 531
Cambridge, Massachusetts 02140
617 661 8900

Contact:
Karen Kosko
Higher Order Software, Inc.
(617) 661-8900
-or-
Paula Neely
Newsome & Company, Inc.
(617) 426-4300

FOR TECHNICAL EDITORS

AN OVERVIEW OF USE.IT

USE.IT is an automated software system development tool which addresses the entire software life-cycle from requirements to maintenance. It can produce reliable systems in any application area, from science to engineering to business.

USE.IT implements the HOS Software Development Methodology, a mathematically-based, universally applicable, automated systems development methodology. The HOS Methodology reduces the incidence of errors and provides the maximum clarity of thought and procedure. It does this by providing and enforcing a rigorous analysis of system logic; by insuring traceability of concept from requirements straight through to code; and by providing significant automation of error-prone and laborious manual processes.

-more-

What Can USE.IT Do?

With USE.IT, system developers can:

- State the requirements, specifications and design of any system in a single language that, while mathematically based, is nevertheless easy for anyone to understand and use.
- Automatically analyze these system descriptions at any stage of development to detect logical errors.
- Run prototypes of these system descriptions at any stage of development to verify that the system is operating as expected.
- Automatically generate computer code that exactly corresponds to the verified specifications.
- Automatically generate English-language documentation for the system.
- Maintain the completed system by making changes to the specification, and re-generating new code to match the changed specification.

These capabilities of USE.IT, and the HOS Methodology itself, are based on a simplified set of three system description structures. These three structures, called JOIN, INCLUDE, and OR, represent the three basic types of processing, i.e., dependent processes, independent processes, and choice of processes.

-more-

In the JOIN (J) structure (Fig. 1), which represents dependent processes, the right hand child of the parent function processes the inputs to the parent function, and passes its outputs to the left hand child. This function processes these inputs and produces the outputs of the parent.

In the INCLUDE (I) structure (Fig. 2), which represents independent processes, each child processes a portion of the inputs to the parent function, and each produces a portion of the outputs of the parent.

In the OR (O) structure (Fig. 3), which represents choice of processes, one of the child processes is chosen, based on a boolean variable, to process the inputs of the parent and produce the outputs of the parent.

The structures, whose rules can be learned by anyone in an hour, can be used to define any functional system. Systems defined using these structures can be machine analyzed for errors at any stage of the development process.

System Specification

During system specification, system functions are decomposed into a general hierarchical tree structure with specific properties. This structure has been proven to be mathematically correct. In this tree structure, complex functions are broken down into smaller, simpler

functions. These functions are in turn broken down into even simpler functions, and so forth.

Data variables (inputs and outputs), are assigned to these functions according to the three rules, depending on the relationships between the functions. The names of both data variables and functions can be suited to the specific application, producing a specification that even non-technical people can understand.

HOS functional decomposition is a logical, simple process producing a system specification that is logically consistent and complete. Errors in specifications can be detected by computer early in the development cycle, when error detection and resolution are up to 100 times less expensive than in the testing phase.

This specification also can serve as the basis for automatic source code generation. This capability has two distinct advantages: first, the elimination of manual production programming; second, the ability, using simulation of incomplete functions, to automatically generate working prototypes at any stage of the development process.

The HOS Functional Life Cycle

The HOS Functional Life Cycle is a complete departure from the familiar "waterfall" life cycle. In this traditional life cycle, requirements, specifications,

-more-

design, and code are manually written in different languages and with different procedures. Errors, both logical and conceptual, are not found until the test stage. Manual programming takes up a major portion of development time. And the maintenance process consumes enormous resources, often contributing to, rather than reducing, the disintegration of the system.

The HOS Functional Life Cycle (Fig. 4) is quite different, consisting of three distinct steps:

Step 1: Create Specifications with the Graphics Editor.

Step 2: Analyze and Prototype-test Specifications (Automatic).

Step 3: Implement Specifications (Automatic).

Step 1: Create Specifications

The systems analyst creates functional specifications in close cooperation with the end user. Using familiar structured design techniques, the analyst decomposes the application into a tree structure. The analyst specifies the tree structure on-line, in graphical format, with the USE.IT Graphics Editor.

The tree structure is specified according to the three simple rules, which enable both data flow and control relationships to be defined in a single diagram. Also, for maximum clarity, function-names and data-variable

names can be specified that are germane to the application being created.

Step 2: Analyze and Prototype-test Specifications

At any point in the specification process, the analyst can invoke the USE.IT Analyzer. The Analyzer automatically detects logical specification errors by performing an exhaustive analysis of all data and control relationships throughout the entire tree structure. Errors consist of violations of the specification rules and include data typing, control, recursion, data conflict, and interface errors. All errors are displayed on the terminal screen.

The analyst then uses the Graphics Editor to correct specification errors. He continues analyzing and editing in an iterative manner until no more errors are found. At this point, the specifications have been proven to be logically consistent and complete.

Also, at any stage of the specification process, the analyst, by directing USE.IT to simulate incompletely defined functions, can generate and run a system or subsystem prototype to test the conceptual correctness of the current specifications.

Prototyping is frequently used to demonstrate the operation of the system to an end-user early in the development process. The ability to quickly create and

run prototypes enables the end-user to participate closely in the specification process and eliminates the problem of systems that do not do what the user wanted them to do.

Step 3: Implement Specifications

When the specifications have been analyzed for logical completeness and consistency, and have been prototype-tested for conceptual correctness, USE.IT will implement those specifications for a specific machine environment.

The USE.IT module which performs this task is called the Resource Allocation Tool (RAT). After the RAT has generated source code, the code is compiled to produce object modules, and the object modules are linked to create run modules.

Maintenance with USE.IT

Maintenance of USE.IT systems is an equally simple procedure; it proceeds in the same manner as development. When changes, either error-corrections or enhancements, are to be made to an existing system, they are made to the specification, using the Graphics Editor. These new specifications are then analyzed (and prototype tested if necessary), and new code is re-generated to match the changed and verified specifications.

For further information contact: Higher Order Software, Inc., 2067 Massachusetts Avenue, Cambridge, MA 02140. Tel: (617) 661-8900.

# # #

4382c/040485/AMD

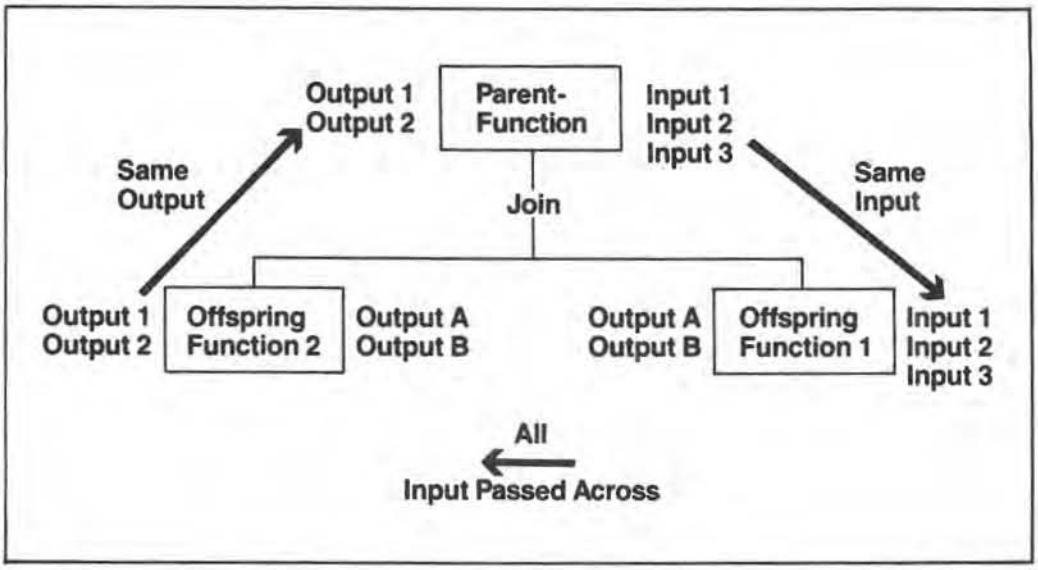


Figure 1

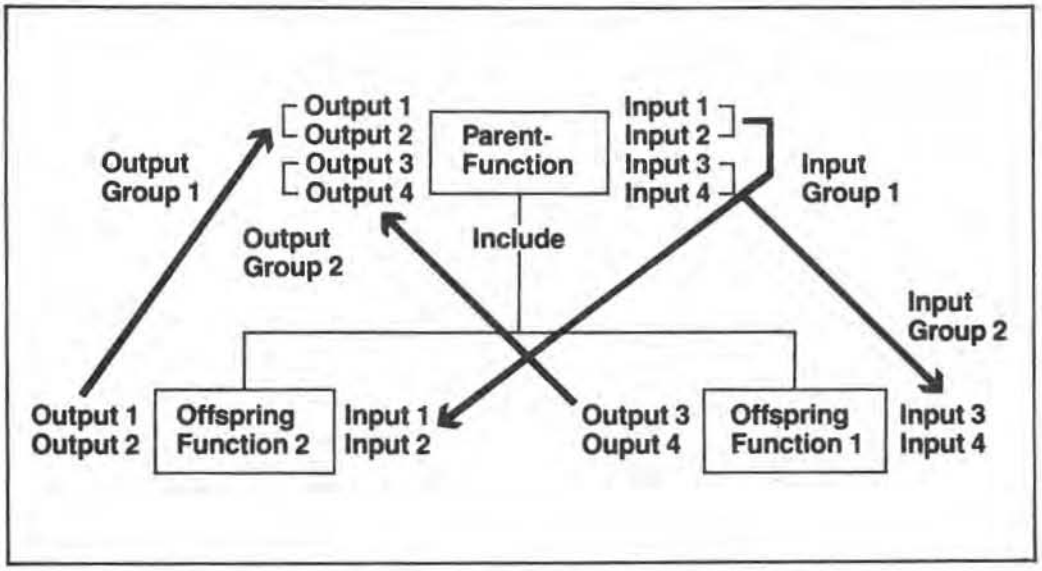


Figure 2

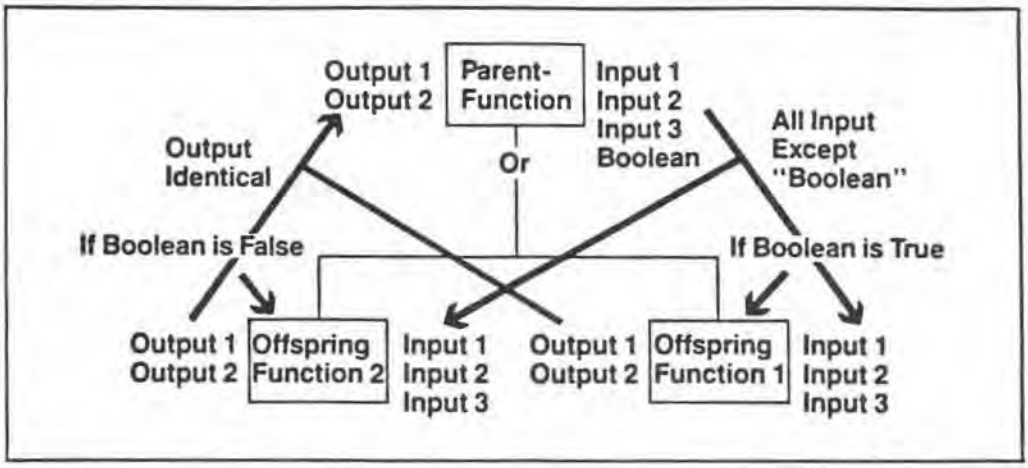


Figure 3

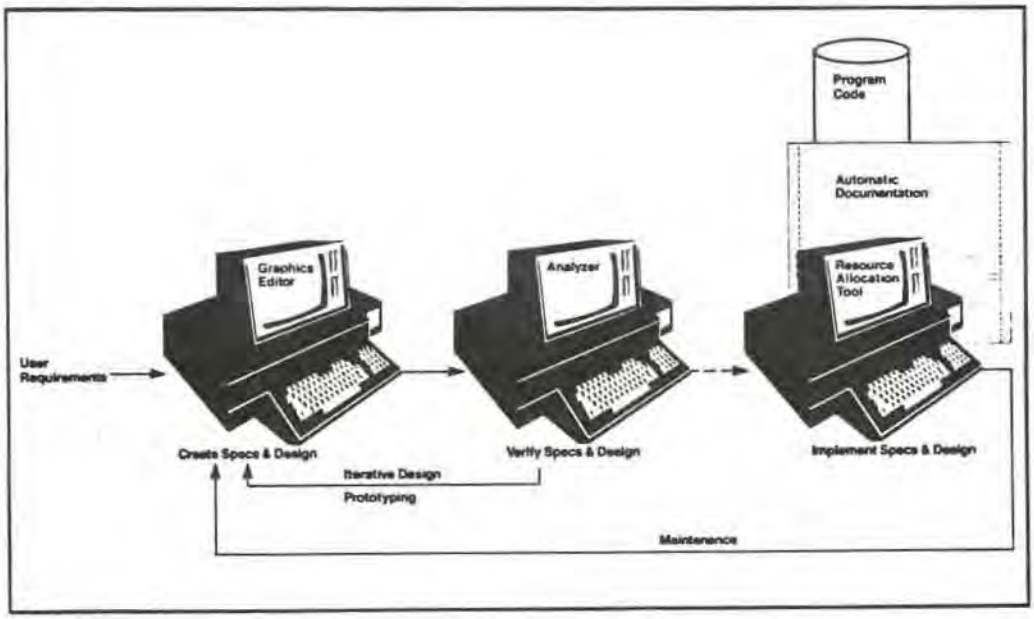


Figure 4

HOS
Higher Order Software, Inc.
2067 Massachusetts Avenue
PO Box 531
Cambridge, Massachusetts 02140
617 661 (illegible)

AUTOMATING SOFTWARE DEVELOPMENT

BACKGROUNDER

Ever since computers were invented, people who use them have been looking for ways to make programming easier. The original binary coding, done manually, was replaced by assembly languages and then by higher-level languages such as BASIC and COBOL. More recently, tools such as program and application generators, database management systems, and screen painters have helped automate the programming phase of software development, addressing the goal of producing more lines of code more quickly. And to a great extent, these tools have succeeded.

Database management systems take over much of the data modeling and data maintenance operations. Fourth-generation languages supported by report and code generators make it possible to produce representations of simple applications quickly and to extend them into useful production systems. Screen painters, documentors, and smart utilities also make specific tasks easier to accomplish.

-more-

International Data Corp. (IDC) estimates that large-scale systems users spent approximately $126 million on design and development tools in 1984, an increase of nearly 50 percent over 1983. IDC predicts that figure will more than double by 1986 and will reach more than $500 million by 1988. Approximately 28 percent of the 1,100 IBM sites polled in another IDC study use an application generator.

Despite the use of these tools, the backlog of applications continues to grow. A survey by Applied Computer Research, Inc. showed that in 1984 large data processing departments had an average applications backlog of more than 27 months. Only three years ago a similar survey found a backlog of only 19 months.

Not only is the backlog growing, but the applications the data processing department is being asked to develop -- insurance claims systems, order processing systems, financial control systems -- also are becoming more complex. Transforming these applications into reliable software is time-consuming and expensive at best, and in some cases impossible, using current development tools.

It is clear that programmer productivity is not the most significant problem facing software developers. Programmer productivity tools have not cut backlogs, and while the programs they help develop may be somewhat more reliable, they address only a small set of applications.

Historically, software development has followed a life cycle which generally includes requirements, specifications, design, implementation, integration, testing, deployment, and maintenance. Requirements and specifications often contain ambiguities. Error-prone manual processes are used within and between phases. Testing is only done near the end, after programming, rather than near the beginning when errors can be caught before they propagate throughout a system. The requirements of the software system are influenced by considerations of the hardware on which it will run and with which tools it will be developed.

Most errors enter the software design process in the specification phase -- and it is very difficult to eradicate these errors once a program is developed. From 50 to 70 percent of data processing budgets are allocated to application maintenance. And it is 10 to 100 times more expensive to correct an error in the testing or maintenance stages than in the requirements or design stages.

To address this problem, a number of structured development techniques attempt to improve the process of formulating requirements and specifications. These techniques make use of data flow diagrams, structure diagrams, and mini-specifications, all developed manually. Also, as many as five languages may be used

before a program is hand-coded, and the interface between each language provides numerous opportunities for misunderstandings and errors. No portion of the development sequence is automated, and there is no way to prove the correctness of data flow diagrams and the corresponding structure charts.

The need is growing for an automated development tool that produces reliable specifications and then automatically generates code and documentation from those specifications.

Higher Order Software, Inc. has developed such a tool, USE.IT, which attacks the root problem of software development -- the traditional life cycle itself. USE.IT implements the life cycle by having the developer start off by defining the specifications in a hierarchical manner. USE.IT then analyzes the developer's definition for consistency and logical completeness. The definition and analysis phases may be repeated several times before a complete set of unambiguous specifications are developed. Finally, USE.IT produces programming code automatically from the specifications. Documentation also is produced automatically.

With USE.IT, software development is automated. Software designers can create specifications that are guaranteed to be logically consistent and complete before a single line of code is produced. USE.IT prescribes a

set of rules that monitor the correctness of any software system automatically. It supports the the entire life-cycle of an application, including:

- Detection of errors in the early stages of the development process, where they are easiest (and cheapest) to fix.

- Integration of the development process by a rigorous and traceable linkage between requirements, design, specification, and implementation.

- Insulation of business knowledge from computer knowledge, by keeping descriptions of what the system does in business terms separate from descriptions of how it is done on the computer.

- Automated prototyping capability, so that the user of the system can actually run the system, and know that it meets his requirements before any final code is written.

- Automated generation of final production code, so that code is guaranteed to match the specification.

- A formal link between the running system and new enhancements required by a changing business environment. With such a link, systems will no longer become degraded in the maintenance stage, and can endure indefinitely.

- Automated generation of system documentation, so that documentation is always current and correct.

- Automated management of reusablility. Re-use of existing programs and structures can significantly reduce both delivery time and cost of new systems. In addition to reusable code and data, USE.IT offers the unique capability of managing reusable specifications and designs -- a quantum step in productivity.

USE.IT has the potential not only to drastically reduce program errors but to attack the problem of ever-escalating application backlogs. Programmers are freed from the burden of maintaining existing systems to work on the new systems management requires. Applications will more closely meet the needs of end-users because specifications are rigorously defined at the beginning of the development cycle.

# # #

AN OVERVIEW OF USE.IT

Higher Order Software, Inc.

HOS

AN OVERVIEW OF USE.IT

WHAT IS USE.IT?

USE.IT is an automated software system development tool that enables developers to produce more reliable systems, and produce them faster than traditional techniques. It is a complete development tool which addresses the entire life-cycle, from requirements to maintenance. It is also completely generalized, and can be used to produce reliable systems in any application area, from science to engineering to business. USE.IT is easy to learn and easy to use, even for non-technical people. It makes the entire process of developing and maintaining software systems more compact, more logical, and more straightforward.

USE.IT implements the HOS Software Development Methodology, which is the first mathematically-based, easy-to-use, universally applicable, and automated systems development methodology to address the entire software life-cycle. The HOS Methodology reduces the incidence of errors and provides the maximum clarity of thought and procedure. It does this by providing and enforcing a rigorous analysis of system logic; by insuring traceability of concept from requirements straight through to code; and by providing significant automation of error-prone and laborious manual processes.

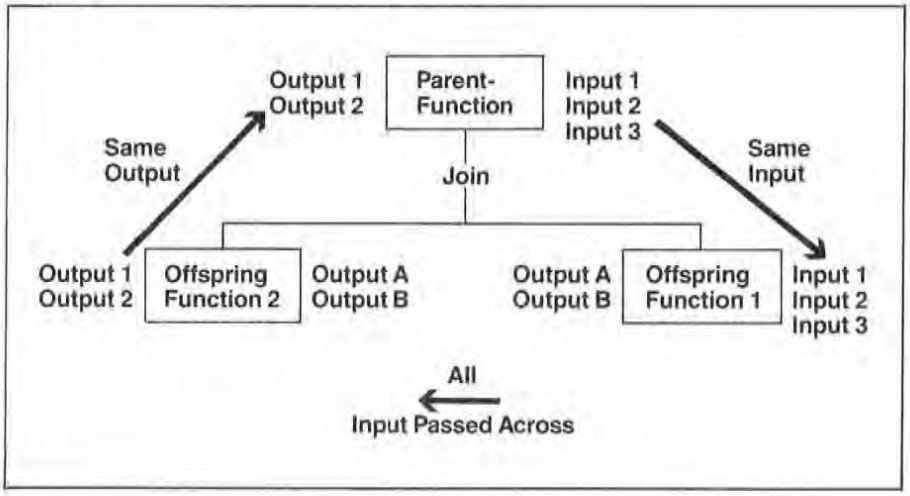


Figure 1

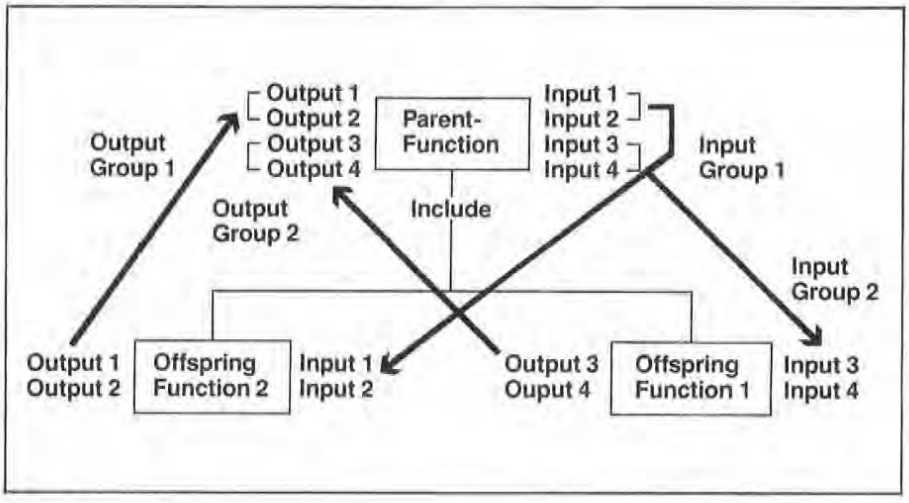


Figure 2

WHAT CAN USE.IT DO?

With USE.IT, system developers can:

State the requirements, specifications, and design of any system in a single language that, while mathematically based, is nevertheless easy for anyone to understand and use.

Automatically analyze these system descriptions at any stage of development to detect syntactical, dataflow, and interface errors.

Run prototypes of these system descriptions at any stage of development to verify that the system is operating as expected.

Automatically generate computer code that exactly corresponds to the verified specifications.

Automatically generate English-language documentation for the system.

Maintain the completed system by making changes to the specification/design, and regenerating new code to match the changed specification.

These capabilities of USE.IT, and the HOS Methodology itself, are based on a unique and simplified set of three system description structures.

THE HOS CONTROL STRUCTURES

The three primitive HOS control structures, called JOIN, INCLUDE, and OR, represent three types of processing which are common to all functional systems. These are: dependent processing, in which one process (function) depends on another process to provide its data; independent processing, in which neither process depends on the other for data; and choice processing, in which one of two processes is chosen to fulfill a function.

In the JOIN (J) structure (See Figure 1), which represents dependent processes, the right hand child of the parent function processes the inputs to the parent function, and passes its outputs to the left hand child. This function processes these inputs and produces the outputs of the parent.

In the INCLUDE (I) structure (See Figure 2), which represents independent processes, each child processes a portion of the inputs to the parent function, and each produces a portion of the outputs of the parent.

In the OR (O) structure (See Figure 3), which represents choice of processes, one of the child processes is chosen, based on a boolean variable, to process the inputs of the parent and produce the outputs of the parent.

The HOS control structures, whose rules can be learned by anyone in an hour, can be used to define any functional system. Systems defined using these structures can be computer-analyzed for errors at any stage of the development process.

SYSTEM SPECIFICATION

When specifying systems, functions are decomposed into a general hierarchical tree structure (See Figure 4), which is a familiar component of the most modern structured design techniques. But the HOS tree structure is unique: it is a specific type of tree structure with specific dataflow and control properties, and it has been proven to be mathematically correct.

In this tree structure, complex functions are broken down into smaller, simpler functions. These functions are in turn broken down into even simpler functions, and so forth. This process of breaking down functions into a hierarchical tree structure is known as functional decomposition.

Data variables (inputs and outputs), are assigned to these functions according to the three simple specification rules, depending on the relationships between the functions. The names of both data variables and functions can be suited to the specific application, producing a specification that is readable by even non-technical people.

HOS functional decomposition is a logical, simple process producing a system specification that is both easy to understand, and logically consistent and complete. And an HOS specification also can be analyzed by computer; errors in specifications can be detected automatically early in the development cycle, when error detection and resolution are up to 100 times less expensive.

But automatic analysis is not the only advantage of a mathematically-based system description. Computer-readable syntax and mathematically rigorous structure allow HOS specifications to serve as the basis for automatic source code generation. This capability has two distinct advantages: first, obviously, is the elimination of manual production programming; the second is the ability to automatically generate working prototypes, at any stage of the development process.

These capabilities, automatic specification analysis and automatic source code generation, have been realized in the USE.IT tool, and the result is, simply, a better way of developing software systems – the HOS Functional Life Cycle.

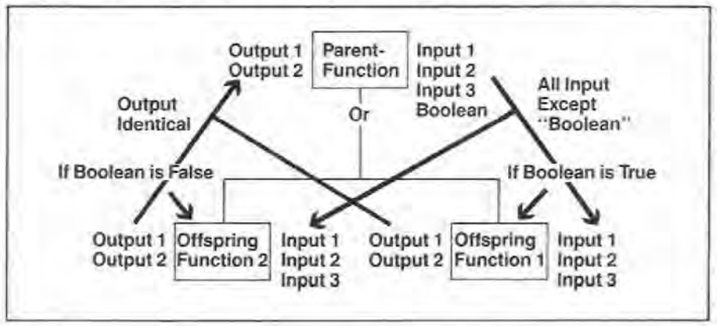


Figure 3

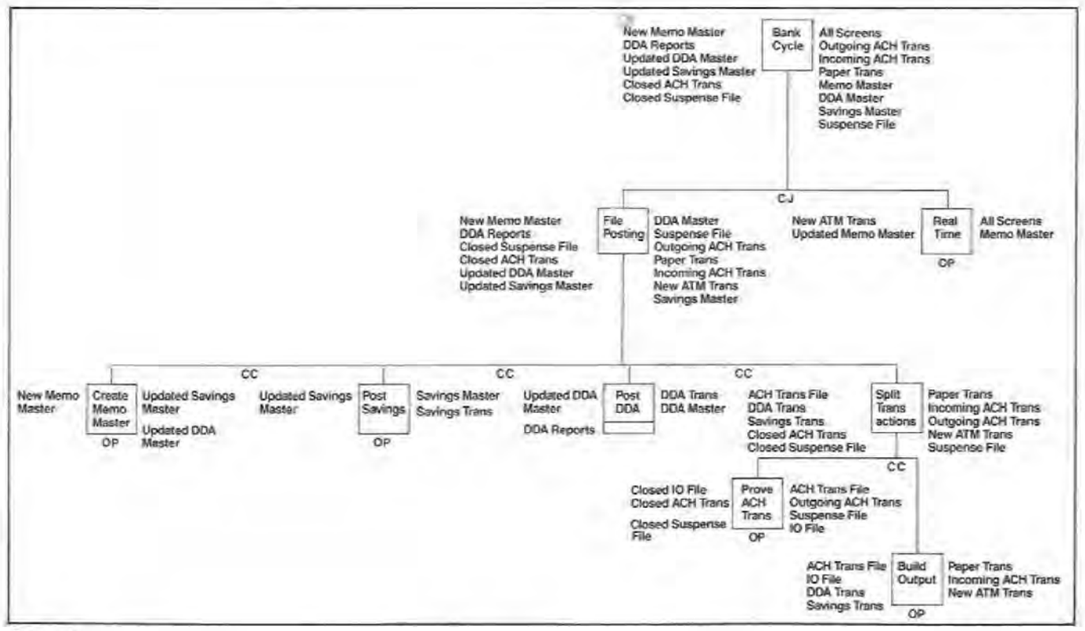


Figure 4

THE HOS FUNCTIONAL LIFE CYCLE

The HOS Functional Life Cycle is a complete departure from the familiar "waterfall" life cycle (See Figure 5). In this traditional life cycle, requirements, specifications, design, and code are written in different languages, and with different procedures, using manual processes. Frequently, errors are not found until the test stage. Programming takes up a major portion of development time. And, the maintenance process consumes enormous resources, and often contributes to, rather than reduces, the disintegration of the system.

The HOS Functional Life Cycle (See Figure 6) is quite different. It is an integrated, logical, automated, and dependable system development process which produces conceptually correct, reliable, and easily maintainable systems.

BUILDING SYSTEMS WITH USE.IT

The procedure for developing systems with USE.IT is remarkably simple. It consists of three distinct steps:

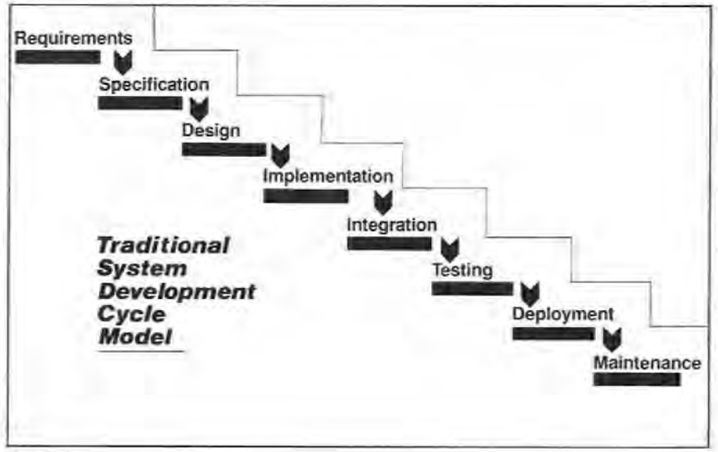


Figure 5

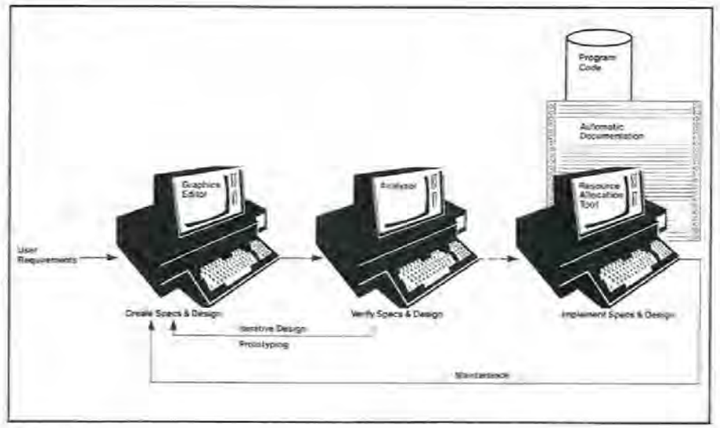


Figure 6

Step 1: Create Specifications.
Iterate
Step 2: Analyze and Prototype-test Specifications (Automatic).
Step 3: Implement Specifications (Automatic).

STEP 1: CREATE SPECIFICATIONS

A complete USE.IT system specification consists of a set of control maps and a set of datatype definitions. The control maps, using the HOS structures, define all the functions performed by the system, the control of those functions, and the flow of data through the functions. The datatype definitions define all the data either used or produced by the system.

CREATING CONTROL MAPS Control maps are created by systems analysts, using the USE.IT Graphics Editor. They construct, in graphical format at the terminal screen, HOS tree structures of functions and data. With the Editor, the analysts can assign names to functions, and decompose those functions into less complex functions. They can also assign input and output data to each function, and specify the control structure appropriate to each functional family.

In the early stages of the development process, the analyst works closely with the end-user in the development of higher-level functional requirements. When these are completed, further decomposition of the system into specifications and design can be performed by a team (or several teams) of analysts, with the guarantee that the interfaces of all modules will be correct.

During the entire specification development process, plots of control maps as well as English-language documentation of the system can be automatically generated, improving communication between members of the development team.

CREATING DATATYPE DEFINITIONS USE.IT provides a library containing many useful general-purpose datatypes. Many systems can be developed using these datatypes alone. In some cases, however, developers may wish to define their own data. This can be done while the control maps for the system are being developed.

A datatype is defined by identifying to USE.IT all the operations (called primitive operations) that can be performed on members of that type. Short, reliable code macros are then written that implement these primitive operations; later, when automatic implementation of the specification takes place, these macros will automatically be inserted in the code at the appropriate places.

STEP 2: ANALYZE AND PROTOTYPE-TEST SPECIFICATIONS

At any time during the system specification process, the analyst can invoke the USE.IT Analyzer to verify the entire system, or any part of it. The Analyzer automatically detects logical specification errors by performing an exhaustive analysis of all data and control relationships throughout the entire tree structure.

By definition, errors consist of violations of the control structures rules. Numerous categories of logical specification errors are detected automatically by the Analyzer, including data typing errors, control errors, recursion errors, data conflict errors, and interface errors.

Figure 7 shows an example of an error that has been detected by the Analyzer. All errors found are displayed on the terminal screen. The analyst then uses the Graphics Editor to correct specification errors. One continues to detect and correct errors in an iterative manner until the specifications have been proved to be logically consistent and complete.

Also, at any stage of the specification process, the analyst, by directing USE.IT to simulate incompletely defined functions, can generate and run a system or sub-system prototype to test the conceptual correctness of the current specifications.

Prototyping is frequently used to demonstrate the operation of the system to an end-user early in the development process. The ability to quickly create and run prototypes enables the end-user to participate closely in the specification process and eliminates the problem of systems that do not do what the user wanted them to do.

Figure 8 is an example of the screen output of USE.IT when running a prototype. As the prototype runs, the user is first asked to enter the test value inputs to the top node of the tree. Completed functions execute as specified; when an incomplete (simulated) function appears (1), the inputs to the function are displayed (2), and the user is asked to enter data to represent the output of the simulated function (3). Execution of the prototype then continues in this way, executing completed functions and simulating incomplete functions, to completion.

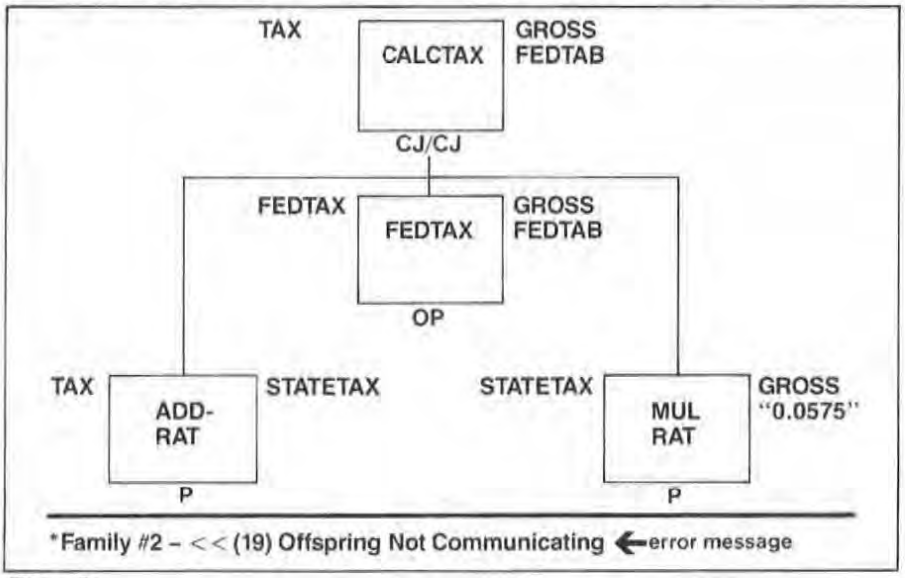


Figure 7

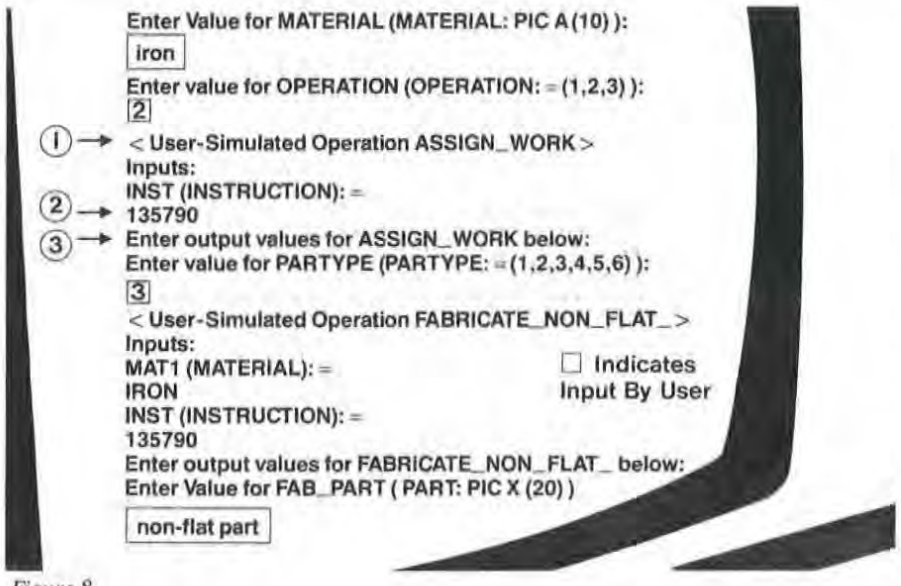


Figure 8

STEP 3: IMPLEMENT SPECIFICATIONS

When the specifications have been analyzed for logical completeness and consistency, and have been prototype-tested for conceptual correctness, the analyst invokes a USE.IT module called the RAT (Resource Allocation Tool) to automatically implement those specifications for a specific machine environment.

After the RAT has read through the entire tree structure and generated source code in the desired language, the source code is compiled to produce object modules. These modules are linked to produce executable code that will run on the computer.

A complete set of control map plots and a complete English-language System Document are then generated, and the system is complete.

MAINTENANCE WITH USE.IT

Maintenance of USE.IT systems is an equally simple procedure; it proceeds in the same manner as development. When changes, either error-corrections or enhancements, are to be made to an existing system, they are made to the specification, using the Graphics Editor. These new specifications are then analyzed (and prototype tested if necessary), and new code is regenerated to match the changed and verified specifications.

SUMMARY

The USE.IT Functional Life Cycle provides improvements in both reliability and production over the familiar software development life cycle.

Improvements in reliability result from improved communication between developers due to the use of a single language. End-users are more involved in the development process, due to the clarity of HOS specifications and to the ability of USE.IT to automatically generate and execute prototypes at any stage in the development process. Specifications are automatically analyzed. Error-prone programming is reduced or eliminated.

Improvements in production result from early detection of errors through analysis and prototype test and the reduction or elimination of manual programming. Existing, tested modules can be interfaced simply to new applications. Automated code generation provides increased portability of functional specifications.

USE.IT is simply a better way to build software systems.

Headquarters
2067 Massachusetts Avenue
Post Office Box 531
Cambridge, Massachusetts 02140
617-661-8900
Telex 951253 HOS INC CAM

Eastern Region
551 Fifth Avenue
Suite 1110
New York, New York 10176-0027
212-490-8721

Western Region
8445 Freeport Parkway
Suite 420
Interfirst Place
Irving, Texas 75063
214-257-3758

Higher Order Software, Inc.

HOS
Higher Order Software, Inc.

HIGHER ORDER SOFTWARE, INC.

BACKGROUND

Margaret H. Hamilton and Saydean Zeldin founded Higher Order Software, Inc. (HOS) in 1976, after Hamilton had managed the entire on-board software development project for the Apollo space mission at M.I.T.'s Draper Laboratory. Zeldin also held a key position in the Apollo effort. Hamilton is president and chief executive officer of HOS, and Zeldin is executive vice president and chief financial officer.

Hamilton and Zeldin developed a rigorous, mathematical theory based on empirical studies they performed on the appolo project. This theory spawned a methodology and commercial software products consistent with that methodology. These products address the $2.5 billions systems software market by providing capabilities formerly thought impossible.

One of the products, USE.IT, is a set of software programs which is used to automate the process of building logically correct software. For the first time, a single tool can be used throughout the life cycle of a project by all team members to specify, generate and maintain logically correct software applications.

-more-

2067 Massachusetts Avenue, Post Office Box 531, Cambridge, Massachusetts 02140 617-661-8900

Hamilton and Zeldin have been joined at HOS by James Frame, executive vice president and chief operations officer. Frame has more than 22 years of software business experience at IBM and for six years was corporate vice president at ITT.

The Cambridge, Mass. company has raised more than $9.2 million in venture capital, $5.7 million as recently as April, 1984. Investors include Alex. Brown & Sons; Cazenove; Emerging Growth; Frontenac; Greylock; Henry & Co.; the Hillman Fund; James Martin; Merrill Lynch; Samuel Montagu Co.; Newcastle; Sears Pension Trust; J.F. Shea Co.; and Venrock. HOS quadrupled revenues in fiscal 1983.

Sales offices across the United States serve a roster of Fortune 200 and other major corporations who have recognized the value of HOS products and services. Computertime Network Corporation, Ltd. of Montreal, P.Q. is a Canadian licensee.

HOS and Sema-Metra, a leading French high technology firm, are involved in a joint venture to market HOS products in Europe.

James Martin, the well-known consultant and author, has devoted a book, "Program Design Which Is Provably Correct," to HOS products and methodology. In the book he said, "This methodology is so powerful that it needs to be regarded as a major new technology for creating systems....The beginnings of true software engineering."

# # #

HOS
Higher Order Software, Inc.

CONTACT:
Thomas D. Lutz
Higher Order Software, Inc.
(617) 661-8900

FOR IMMEDIATE RELEASE

HIGHER ORDER SOFTWARE APPOINTS JAMES FRAME
EXECUTIVE VICE PRESIDENT AND CHIEF OPERATING OFFICER

CAMBRIDGE, Mass., Oct. 4, 1984 -- James Frame has been appointed executive vice president and chief operating officer of Higher Order Software, Inc. (HOS), the company announced today. Frame brings 28 years of business experience at IBM and ITT to his new position.

HOS creates, markets and services automated software development tools, such as USE.IT for the Digital Equipment Corp. VAX line of superminicomputers. These tools dramatically increase the productivity of large software application design and maintenance.

"The addition of Jim to our management team uniquely positions HOS for the introduction of a significant line of software products to the IBM marketplace," said Margaret H. Hamilton, president and chief executive officer of HOS. "During his 22 years of top management achievement at IBM, he pioneered the development of software as a standalone business product. With Jim's help, we can extend our industry leadership in

-more-

providing the highest level of business support to executives seeking a competitive edge in the '80's."

Frame said, "HOS products represent a true watershed in the software industry. They have the same potential to dominate the software engineering marketplace for decades to come as IMS has dominated the database marketplace in the past."

Frame joins HOS after six years with ITT, where he was corporate vice president of programming. Under his leadership, the 8,000 software engineers in ITT recorded a dramatic productivity rise within three years.

"The very success of our productivity experiences at ITT clearly demonstrated the need for further, geometrical productivity increases in the future," Frame said. "I am highly confident that HOS can break the vicious computer application backlogs facing the industry."

Previously, Frame spent 22 years at IBM, where he led the business development of many IBM products and services that are industry standards today. These include:

- IMS database management system
- CICS (Customer Information Control System)
- COBOL, PL/1, FORTRAN, APL and BASIC programming languages
- BTAM and TCAM telecommunication access methods
- DOS/VS operating system for the IBM System/370

In addition, Frame established the Programming Center Development Laboratory in Research Triangle Park, North Carolina, and the Santa Teresa Laboratory of San Jose, California.

"Santa Teresa was designed from top to bottom to further software engineering," said Frame. "The Laboratory is the first of its kind in the world, a unique synthesis of architectural beauty, human ergonomics and IBM software production values." It has won numerous awards, including an American Institute of Architects' National Honor Award, and has been imitated by several major corporations.

Frame is a graduate of St. John's College, Annapolis, Maryland, where he is vice chairman of the Board of Visitors and Governors.

# # #

2728c/100384/GF

HOS
Higher Order Software, Inc.

CONTACT:
James Frame
Higher Order Software, Inc.
(617) 661-8900

FOR IMMEDIATE RELEASE

HIGHER ORDER SOFTWARE APPOINTS THOMAS D. LUTZ
VICE PRESIDENT OF MARKETING AND SALES

CAMBRIDGE, Mass., Oct. 4, 1984 -- Higher Order Software, Inc. (HOS) has appointed Thomas D. Lutz as vice president of marketing and sales, the company announced today.

"Tom's unbroken string of successes in the management and systems education business will further strengthen our commitment to professionally market and support our HOS products," said Margaret H. Hamilton, president and chief executive officer of HOS.

Hamilton said, "We are determined to hire the top people throughout the information systems industry; people who combine a reputation for business innovation with an extraordinary depth of information industry experience. Tom not only meets but exceeds these conditions."

Prior to joining HOS, Lutz was principal and director of education and communications at Nolan, Norton and Company, a high technology management consulting firm. He

-more-

Lutz also spent seven years as the head of information systems for the Mayo Foundation, where he was responsible for developing and marketing all clinical, research and administrative systems.

Lutz began his career at IBM as an applied scientist and manager at the Systems Research Institute. During his 14 years with IBM, he founded and directed the IBM Systems Science Institute.

Throughout his career, Lutz has served as an Adjunct Professor to several graduate schools, including the University of Minnesota, the Pratt Institute and the University of Newcastle (U.K.). He has lectured extensively in Asia, Africa, Australia, Europe and South America. He is also the author of several DELTAK video journals on systems management.

Lutz holds a bachelor of science degree in mathematics from South Dakota School of Mines and Technology and an masters of science degree in operations research from New York University.

# # #

2729c/100384/GF

HTI

Hamilton Technologies, Inc.
17 Inman Street
Cambridge, Massachusetts 02139
(617) 492-0058

May 4, 1988

Esther Dyson, President
EDventure Holdings, Inc.
375 Park Avenue
New York, New York 10152

It was a pleasure to talk to you at the Meta Software simulator meeting. As a result of our discussion I am enclosing some information on our company and our product, 001™.

Sincerely,

Margaret Hamilton mhh

Margaret H. Hamilton
President

MHH/nrk
Enclosures

TOWARDS ULTRA RELIABLE MEDICAL SYSTEMS

Margaret H. Hamilton
Hamilton Technologies, Inc.
17 Inman Street
Cambridge, Massachusetts 02139

Abstract

With today's conventional system development techniques, as size and complexity increase so does the probability that a system, when introduced into operation, cannot be trusted. This is despite an inordinate amount of testing and evaluation. And when a system works, the cost of attaining such a state is often needlessly high. The predictable result is wasted dollars, lost time and missed deadlines. For many systems, coping with events that cannot be entirely predicted is vital to effective real-time system performance. The uncertainty of actual environmental conditions at the moment of truth can present challenges to operational reliability that border on the impossible. Such was the case with the Therac 25 radiation therapy environment [1, 2]. As a result of hardware, software and humanware system defects and defects in integrating these systems during development and in real time, people unnecessarily lost their lives. These incidents are a cruel reminder that "One small break in the chain of care can have grave consequences" [3].

For a medical environment, whether it be one with direct or indirect human involvement, a technology which reverses these trends is needed to develop ultra-reliable systems. Ideally, a system that is ultra-reliable has zero defects. Zero-defect systems are theoretically possible, but difficult, to achieve. There are today, however, substantial numbers of errors which exist, or potentially exist, in developed systems or systems to be developed which can be eliminated. This can be accomplished by using a combination of common sense and advanced modeling, simulation and software development techniques.

Properties of Zero-Defect Systems

A *system* is an assemblage of objects united by some form of regular interaction or interdependence. It could consist of hardware, software or humanware objects; or, it could be a combination of any of these. Thus, a person, a computer, a software program or the integration of these objects is a system. A zero-defect system is defined in terms of properties about the system (e.g., its developmental states of existence such as a definition or an implementation, each of which is an evolving input object to the system which develops it) and in terms of properties of the system for its operational states of existence. A *zero-defect system* is one which is reliable in both a formal and practical sense.

A *formal* system is consistent and logically complete; it has no interface errors (or ambiguities). A *practical* system is developed on time and it is affordable to build and operate; it works. A system which *works* will handle the unpredictable, both as a system being developed and as a system being operated; it satisfies the developer's intent; it satisfies the user's intent; it always gets the right answer at the right time and in the right place; it is efficient to operate in time and in space.

To *handle the unpredictable*, a system, during its own development, will handle changing development requirements without affecting unintended areas; it will handle change and the unexpected during its operation. This includes having the ability to reconfigure in real-time, detect errors and recover from them and the ability to be simulated in, operate in, respond to and interface with a distributed, asynchronous, real-time environment.

An *affordable* system has properties which prevent errors from being created in the future. It is portable, flexible, understandable and repeatable; its development requires minimum people time and minimum calendar time.

A *portable* system can be implemented in or operational in different, changing and diverse parallel environments; it can exist in different, changing, diverse, secure and multi-layers of abstraction; it allows for the plug-in or reconfiguration of different modules, or parts of modules, for those objects which can vary in functionality from state to state; it has the ability to be used by various applications and execute on various operational environments (e.g., human, robot and computer environments).

A *flexible* system has the ability for its definition to be changed from many objects to one (providing for abstraction, integration and applicative operators) or from one object to many (providing for decomposition, modularity and computability), as necessary both during its developmental and operational states.

For a system to be *understandable*, one is able to define the integration of all of its objects; trace it and any object in that system (including its behavior and its structure), throughout each phase of development and from one phase to the next; define it to be as simple as possible, but not simpler; define it in such a way that it naturally corresponds to the real world of which it is a model; communicate it with a common semantics to all entities including all levels of users, all levels of developers, all levels of managers and all levels of computing facilities and their respective environments; and one is able to define it with "friendly" definitions (where "friendly" is a relative term with respect to each kind of user), using variable, user selected syntaxes, relating to and being derived from a common semantic base, and capitalizing on the ability to hide unnecessary detail.

A *repeatable* , or reusable, system is defined with mechanisms which inherently facilitiate the process of standardization and the ability to define and use more abstract and common mechanisms, all of which by their very nature support functionally natural modularity (e.g., mechanisms are "dumb" in that they are not aware of nor do they need to be aware of their context of use or their implementations and an object always exists as an integrated entity with respect to structure, behavior and properties of control); to be repeatable a system must inherently provide properties for mechanization (and thus the automation) of its own development processes.

Philosophy

A zero-defect system begins with a set of reliable thoughts which result in a reliable model. A *model* is a tentative definition of a system or theory that accounts for all of its known properties [4]. A model could be defined for just about anything: an airplane, building an airplane, flying an airplane, eating a sandwich, a missile system, planning your day's activities, a patient, a doctor, the process of providing radiation treatment to a patient, a radiation machine or the process of building a radiation machine. A model can be simulated with software. A *simulation* is the "running", exercising, testing or execution of a model. A *software simulation* is the set of instructions (software) which "runs" a simulation on a particular computer.

Once a reliable model is defined (or an unreliable model is redefined to be reliable) a software implementation consistent with the model (i.e., a reliable simulation of the model) is developed. The next step is to build the real system (e.g., a machine if the system is a hardware system). If the real system to be developed is a software system, this step may not be necessary, since a software system already exists as a result of implementing the model. The responsibility for developing a reliable system resides with the user and the developer. The user is responsible for knowing what he wants; the developer is responsible for communicating the user wishes to the computing environment.

The ideal modeling environment begins with reliable building blocks. To build a reliable system, only reliable systems are used as building blocks and only reliable mechanisms (systems, themselves) are used to integrate these building blocks. Each new system, constructed from only reliable systems, is then used along with the more primitive systems to build new, larger and more comprehensive reliable systems.

The philosophy that reliable systems are defined in terms of reliable systems is applied in a more global sense in the management

of a project. For example, if an existing process works or an existing object is correct and an additional equivalent functionality is needed, one should use that existing system and not build another one. Unnecesary extra effort should be avoided; each new effort takes time, introduces new errors and costs money. There are also hidden ways to repeat unnecessary efforts. A system that is not portable has to be implemented over again in order to operate in a new environment; most of such an implementation can be avoided before the fact. If a system is not traceable, much wasted effort is made to find and fix the errors caused by a change, resulting in new errors and wasted time. If a system definition is ambiguous, errors are introduced throughout its evolution and its associated implementations as a result of misunderstandings.

There are times when something should not be repeated or used over again; when something doesn't work, a new effort is justified and a new process begins. Once the new process is well understood, it can be mechanized. Once it is mechanized, it can be automated.

Hand-in-hand with each zero-defect system is the technology (or integrated set of techniques) that was used to produce that system. Measurement standards are needed to determine the *degree* to which a technology approaches zero-defect systems. For example, to what degree does a technology follow the philosophy of building reliable systems in terms of previously existing reliable systems, define a system which is understandable, define every aspect of a system, support abstraction, flexibility and repeatability, reduce unnecessary complexity, allow a system to respond to the unexpected, integrate all aspects of a system definition and its development; and provide automation, systems which are consistent and logically complete and systems that work?

The 001 Technology

One technology, 001™, began in 1968 as part of an effort to finc more reliable methods to develop large, real-time systems within which software resided. At that time an empirical study was performed by Hamilton and her staff on the APOLLO on-board flight software system. The result was a theory [5, 6] which provided the beginnings to many aspects of the 001 technology. From this theory evolved a technology over the last two decades [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. This technology has been automated and put into practical use.

The 001 technology is an integrated hierarchical functional and object-oriented modeling approach based upon a concept of control. Each system is defined in terms of a *control map* (a tree structure whose objects are related in terms of control) with 001 AXES, a language which emphasizes completeness of control over all objects within a system and its environment. The necessary elements to completely specify a system in terms of control appear on a control map. The foundations of 001 AXES are based on a set of six control axioms derived from empirical data of large systems and their developments [5] and on the assumption of the existence of a universal set of objects. Each axiom describes a relation of immediate domination with respect to each node in a hierarchy. The union of these relations is control. The control axioms establish the relationships of objects for invocation, input and output, input access rights, output access rights, error detection and recovery, and ordering. Control affects an object, the relationships of an object, and the development of an object in terms of its relationships. Not only is every object in a system controlled, but every object has a unique controller. Each system is defined with mechanisms derived from the control axioms; the result is a system free of interface errors (or ambiguities) and side effects. Ambiguity is eliminated in understanding the behavior (and structure) of an object and the behavior of that object in terms of its relationships.

A system defined in terms of the control axioms results in many other interesting properties. Modularity that is safe and secure [17] is inherent. (This property ensures that the necessary aspects of a functional specification are treated as an integrated whole and not artificially separated for the sake of superficial modularity; for such a separation can introduce more errors into a system. For example, at any node in a hierarchy, an object can always be identified with respect to its structure, its behavior, its relationships in development and in real-time and with respect to an integrated set of aspects of control.) The definition of the behavior of an object is separate from the definition that uses the object; the definition of a development layer is independent from those layers that evolve from it (for example, the specification of a system or object in a system is independent of its implementation); the definition of a control mechanism specifies total ordering among functions, allowing for the description of that definition to be order independent; both the mechanisms which are defined and the systems defined with these mechanisms behave as if they are "instructions"; (e.g., a given control structure has no knowledge about a higher-level control structure); control, or the chain of command, can be traced directly on a control map. As a result, function flow (including both input objects and output objects) can be traced directly. Communication of functions always takes place at the same level in a hierarchy; changes can be traced and changes can be made locally; concurrent patterns can be automatically detected and traced; the single-reference, single-assignment property of 001 systems provides for an important set of resource allocation alternatives, particularly when it comes to considerations of parallel processing.

A 001 system is defined in terms of a single control map which integrates all other control maps. A functional control map (FMap™) defines a hierarchy of functions; a Type control map (TMap™) defines a hierarchy of abstract types. For each control map definition, three kinds of building blocks are used to model and build *any* kind of system: reliable data types (in order to identify objects), reliable functions (in order to relate objects of types) and reliable control structures (in order to relate functions). These building blocks are used to connect functions to objects, functions to functions and objects to objects (see Figure 1):

Figure 1. A Use of the 3 Basic Mechanisms.

A *control structure* is a mechanism which relates a parent and its offspring according to a set of rules which have been derived from the control axioms. On a functional hierarchy a control structure relates functions (members of data type function). On a type hierarchy a control structure, in the form of a parameterized type, relates Types.

A *function* has a mapping where each input corresponds to one and only one output. One or more objects serve as the input and one or more objects serve as the output. With 001 AXES, a function also includes additional properties, including those relating to access rights and priorities in terms of a concept of control. On a functional hierarchy there is a function at each node.

A *data type* is a particular set of values. A data type definition consists of a set of primitive operations and a set of rules, or axioms; a data type is defined in terms of primitive operations performed on members of that type and on members of other types within its control. Axioms for each data type define the behavior of these members in terms of the relationships between the primitive operations. More abstract types can be defined in terms of more primitive types. A data type could be anything. It could, for example, be a building, an integer, a doctor, a boolean, a radiation machine or radiation, itself. On a type hierarchy there is a type at each node.

The Primitive Control Structures

All 001 models are ultimately defined in terms of three primitive control structures which have been directly derived from the control axioms. The primitive control structures identify control schemata on sets of objects. There is one for dependent relationships, one for independent relationships and one for decision-making relationships.

The use of the three primitive structures are illustrated with an example of an FMap in Figure 2a. An FMap defines actions in terms of relations between states of objects; it can be used to define, integrate and control objects defined by a TMap. Although system,

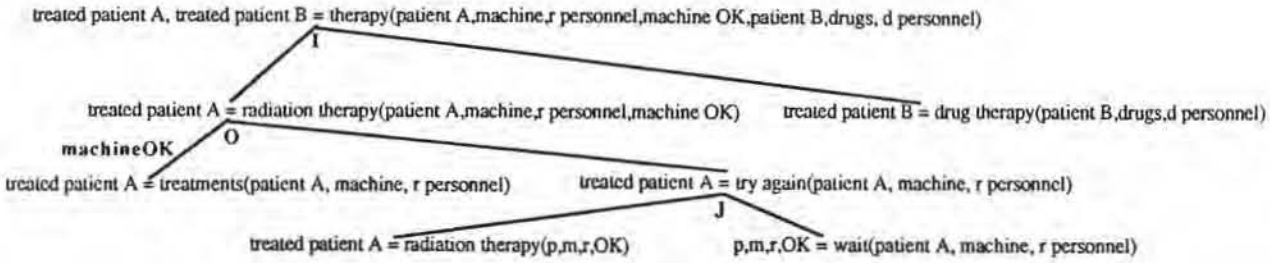


Figure 2a. An Overview of System Therapy Control Flow Oriented Graphics

therapy, is shown in Figure 2a in terms of a tree graphics format which emphasizes control flow, the 001 technology does not dictate a particular syntactical form. This same system, for example, can be defined in textual form. Or, this system can be defined in some other graphical form, if desired, (see, for example, system, *therapy*, defined with controlled data flow graphics, which emphasizes data flow, in Figure 2c) as long as the semantic rules of the technology are adhered to. In all these forms the same information is presented, but that which is highlighted varies from form to form.

In system, *therapy*, the parent function, *therapy*, is decomposed into two offspring functions, *radiation therapy* and *drug therapy*. The function, *therapy*, takes in as input *patient A*, *a machine*, *radiation personnel*, an indicator to check machine status, patient B, drugs and drug therapy personnel. Function,*therapy*, produces treated patients, A and B, as output. Since both *drug therapy* and *radiation therapy* are controlled by *therapy* as independent functions, each takes in its own input from its parent. Thus, for example, the *drug therapy* function takes in its own input (*patient B*, *drugs* and *d personnel*) directly from its parent and produces its own output, *treated patient B*, giving it directly to its parent. *Radiation therapy* likewise takes in its own input directly from its parent and produces its own output. In this relationship between the parent and its offspring, the offspring do not communicate with each other.

The relationship between the *radiation therapy* function and its offspring is one of making a decision. Here, either the *treatments* function will be performed or the *try again* function will be performed. The decision as to which function is to be performed is dependent on the condition of the radiation machine. If the machine is OK, treatments will be performed; if not, try again will be performed. Note, here, that each offspring takes in the same input and produces the same output, since only one of them will be performed for a given performance pass.

The function, *try again*, is decomposed into functions, *radiation therapy* and*wait*. In this case,*try again* controls its offspring in a dependency relationship where radiation therapy depends on *wait*'s output (*p*,*m*,*r*, *OK*), as its input. Here *radiation therapy* cannot complete its performance without *wait*; once it does, *radiation therapy* provides the output,*treated patient A*, for its parent, *try again*.

Each of the three primitive structures has a formal name and a set of rules associated with it for its use (See Figures 3a-3c). The *Join* (J) is for defining the relationships between dependent functions; the *Include* (I) is for defining relationships between independent functions and the *Or* (O) is for defining relationships between decision making functions.

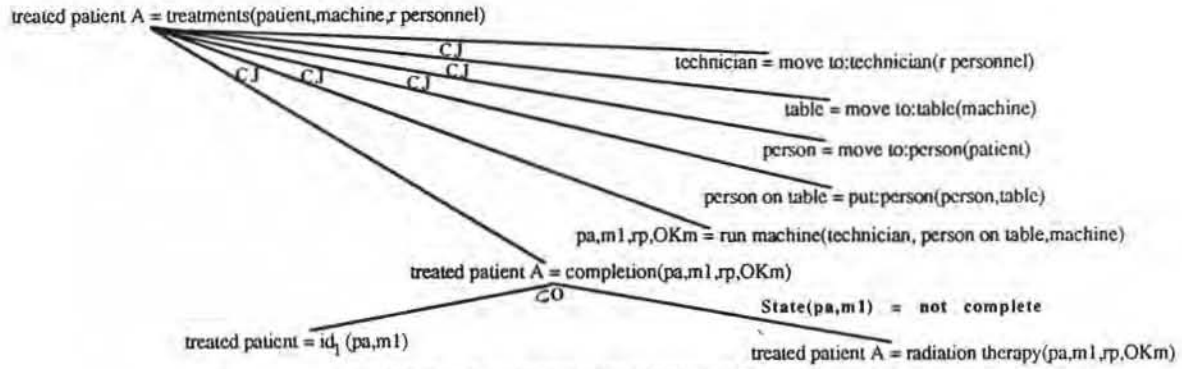


Figure 2b. A Further Decomposition of Therapy.

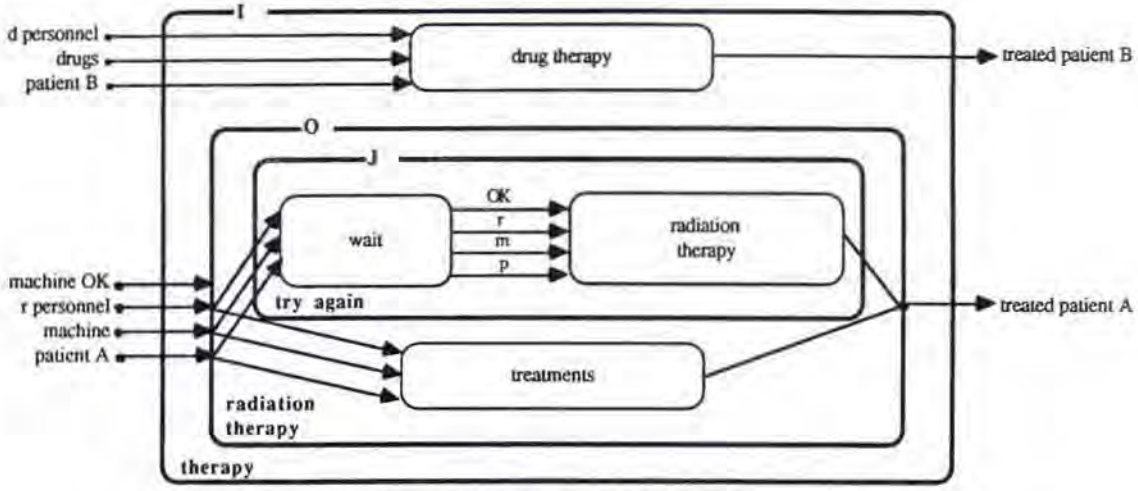
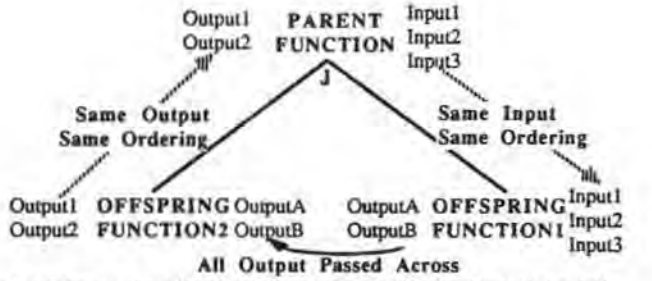
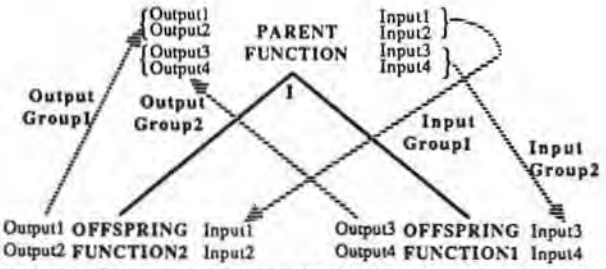


Figure 2c. An Overview of System Therapy Depicted with Controlled Data Flow Oriented Graphics.



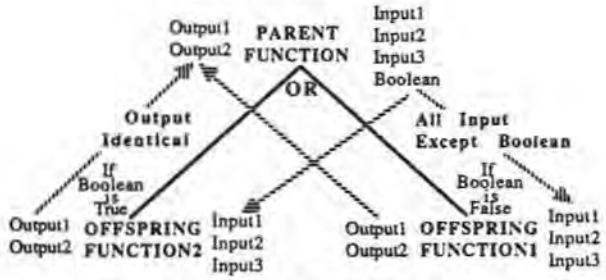
- Inputs to parent are identical to inputs to right offspring (including order)
- Outputs of parent are identical to outputs of left offspring (including order)
- Outputs of right child are identical to inputs of left offspring (including order)

Figure 3a. Syntax Rules Governing JOIN Control Structure.



- The inputs of the parent are inputs to the offspring. Order is maintained; all inputs are used with no overlaps, and no criss-crossing of inputs that are being passed to different children.
- The inputs of the left offspring are identical (including order) to the first inputs of the parent.
- The inputs of the right offspring are identical (including order) to the last inputs of the parent.
- The inputs of the offspring exhaust the parent's inputs.
- The inputs of the offspring are exclusive of each other.
- The outputs of the left offspring are identical (including order) to the first outputs of the parent.
- The outputs of the right offspring are identical to the last outputs of the parent.
- The outputs of the children exhaust the parent's outputs.
- The outputs of the offspring are exclusive of each other.

Figure 3b. Syntax Rules Governing INCLUDE Control Structure.

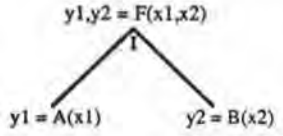


- There must be at least two inputs to the parent node.*
- A boolean value is the last entry on the parent's input list and is input to the parent only.
- The input to both offspring is identical to that of the parent with the exception that the boolean value is not included on the input list.
- The output from both offspring is identical to the parent's output.
- The order of the variables in the offspring's input and output lists must be the same as that of the parent.

*Current implementation of the OR structure.

Figure 3c. Syntax Rules Governing OR Control Structure.

If the rules of the primitive structures are followed, when defining a system, interface errors are eliminated. Interface errors account for at least 75% of the errors found in a large system development [18]. Each of the primitive structures has properties of asynchronous real-time distributed behavior inherent within it. Each system, defined with these structures, is an event interrupt driven system. The existence, therefore, of an event can cause a system to automatically reconfigure and execute a higher priority function. Each function in a hierarchy has a unique priority. A priority precedence is established by the modeler and upheld throughout a given system. For example, in the example below, the left offspring in the *Include* structure always has a higher priority than the right offspring. In this case (Figure 4), given two processors, *A* and *B* can independently begin when *x1* and *x2* become available; failure



Where $x_1 \equiv [x_1]$, $x \equiv [x_2]$ ... and Where x1, x2, y1, y2 each are the children of an input or output Type (i.e., the children of Type, an Ordered Set of Any).

Figure 4. The Priority of A>B.

of *A*'s processor forces an interrupt of *B* 's processor if *B* is processing; if not, then if *x2* becomes available, then *B* must wait until A is finished processing. Given one processor, *A* can initiate when x1 becomes available; *B* can initiate when *x2* becomes available; if both *x1* and *x2* become available simultaneously, then *A* initiates before *B*; *A* can starve *B* of its resources; and *B* can execute before *A* if resources and *x2* are available. Primitive functions are available for activation when all of their input events are available. An abstract function has a lifetime that contains the lifetimes of its immediate children. The primitive control structures relate objects (including functions), each of which has an event state which can be either "past," "present" or "future". The "past" state represents an object that was once alive but is no longer; the "present" state represents an active object; and the "future" state represents an object that might exist or become active. The event state provides a convenient mechanism to understand fully the timing behavior of an object; it also provides a convenient mechanism to trigger an interrupt when a state change of an object occurs.

In addition to illustrating the use of the three primitive control structures, system, *therapy*, in Figure 2a, also illustrates the use of recursion as it is defined in a control map. In system, *therapy*, the function, radiation *therapy*, directly under *therapy*, controls the function, *try again*, which in turn controls *radiation therapy*. This is a shorthand notation for indicating that the lowest *radiation therapy* in the hierarchy represents a repeat of the same pattern as its ancestor (i.e., it controls *try again* and *try again* once again controls a lower level *radiation therapy*, etc., just like its ancestors). Recursion is simply a repetition of the map inside of itself. This definition allows the performance of as many attempts at *radiation therapy* as the machine makes necessary by its malfunctioning.

The same three primitive control structures that were used to define, integrate and control actions in the form of an FMap can be used to define, integrate and control objects in a TMap. Whereas a complete FMap is one whose lowest level nodes have primitive operations on types, a complete TMap is one whose lowest level nodes have primitive types. The type of medical database that might be associated with the FMap in Figure 2 is decomposed into data type *people* and data type *materials* (Figure 5). Here, *people* are defined to be independent of *materials*. *Materials* are shown to be either *drugs* or a *radiation machine*. *Radiation machine* is decomposed into two types of procedures, one of which is dependent on the other.

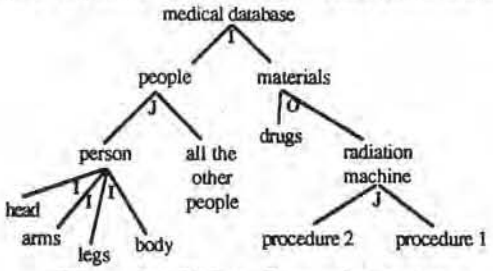
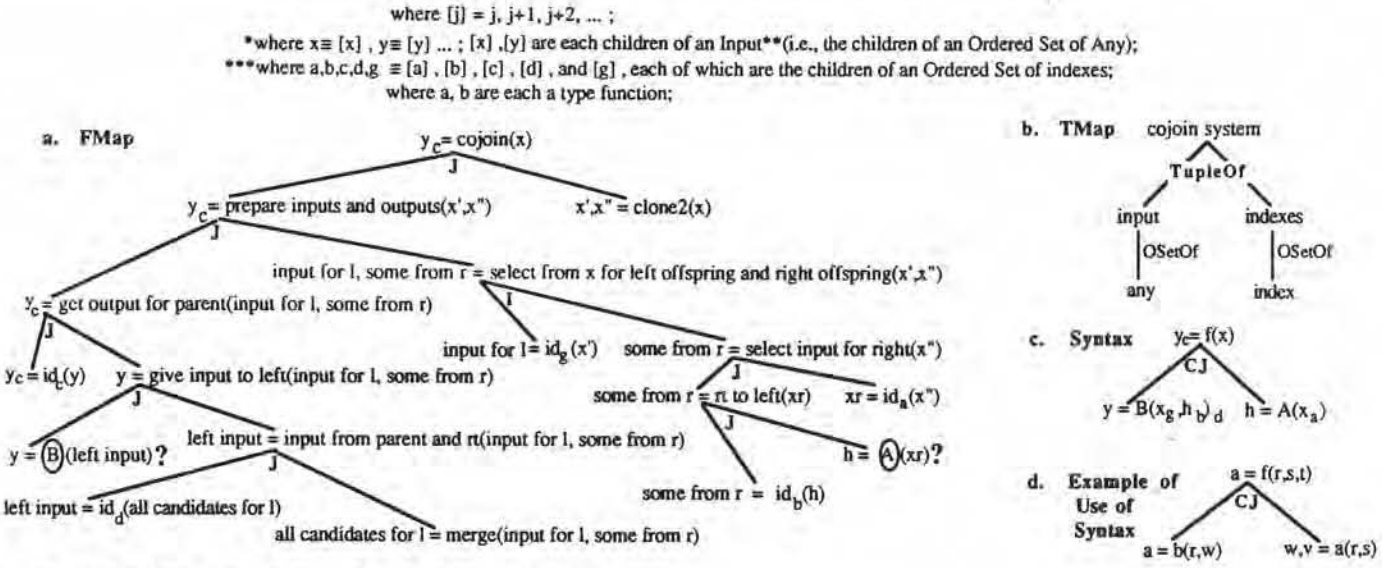


Figure 5. A TMap for Type, *medical database*.

Defined Structures

All FMaps and TMaps could be defined in terms of primitive structures, but there is often a desire to use less primitive structures to accelerate the process of defining and understanding a system. New, more abstract control structures are defined in terms of the primitive structures or in terms of other nonprimitive control structures. The system, *Cojoin*, defined in Figures 6a and 6b, is an example of a system that can be used as a nonprimitive control structure. Among others, this kind of system pattern happens often when using all primitive structures. Within this pattern only the functions *A* and *B* change. The FMap pattern in 6a was defined with primitive structures, *Include* (I) and *Join* (J). The TMap in 6b was defined with nonprimitive structures, themselves defined in terms of primitive structures. As a result of the existence of repeated patterns, the concept of *defined structures* (the ability to define non-primitive structures in terms of more primitive structures) was created. A defined structure allows the user to only show explicitly those variables in the definition which are subject to change with each use of a common pattern. Included with each structure definition is the definition of the syntax for its use (see Figure 6c). Its use (see Figure



Note 1: See definition of id_n and clone2 in [6] and definition of merge in [14].
Note 2: Any and Index are primitive Types.
*All input and output variables are each children of an Ordered Set of Any;
**Input, here, could mean Input or Output.
***All subscripts are each children of an Ordered Set of index.

Figure 6. A Definition for the Cojoin Defined Structure.

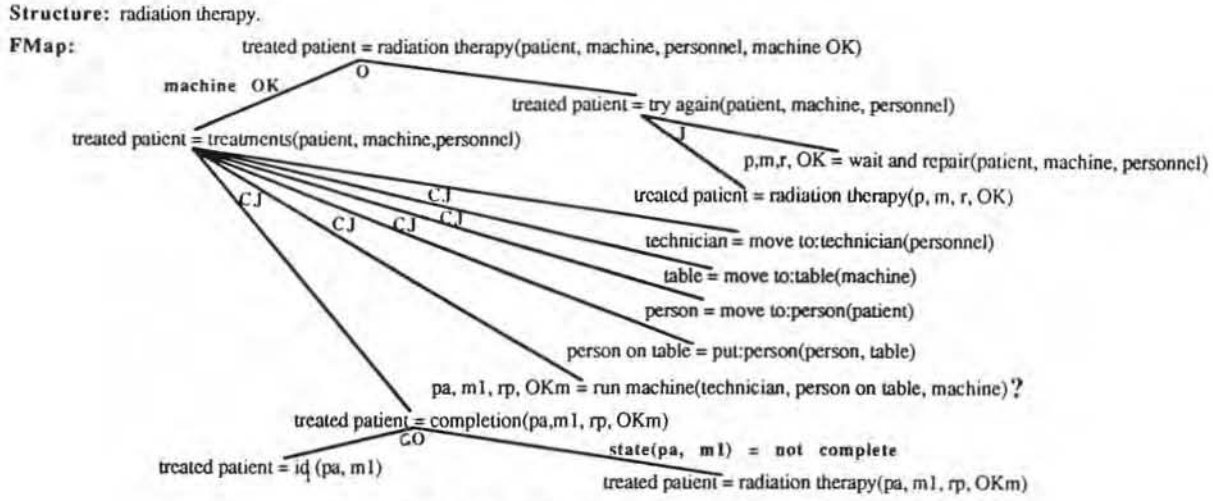


Figure 7. A Defined Structure for Performing Radiation Therapy.

6d) will provide a "hidden repeat" of the entire system as defined, but explicitly show only the necesary elements for defining functions *A* and *B*.

To illustrate further the definition and use of a defined structure, the function, *treatments*, in Figure 2a is further decomposed (Figure 2b) into several functions, starting with *get:technician* on the right hand side and ending with *completion* on the left-hand side. In this decomposition the defined structure, *Cojoin* (CJ), is used several times, in an N-ary structure, to define the relationship between the parent function, *treatments* , and its offspring. The lower levels of system, *therapy*, (see *treatments* in Figure 2b) complete part of the definition of system, *therapy*, since this part of the system was decomposed until primitive functions on previously defined data types were used (see under *treatments*, for example, Id_1, or *get:table*, which are primitive operations on data type, *any*, and data type, *machine*, respectively) or until a recursive function (see again *radiation therapy* whose ancestor is near the top of the same system in Figure 2a). If an operation at the bottom is not primitive, it can be further decomposed (see function, *treatments*, in Figure 2a which is further decomposed in figure 2b) or it can refer to an existing operation in the library. External operations from outside environments can also be used as operations. Operations are defined implicitly by deriving them mathematically from the axioms on a type or explicitly in terms of control structures using already defined operations on a type. When an operation is defined both implicitly and explicitly, the intent of the specification can be cross-checked for correctness.

The *radiation therapy* portion of the *therapy* system can be made into a defined structure (Figure 7). Here the semantics of the definition is the same as the subsystem definition of *radiation therapy* in system, *therapy*, except the run machine node is variable. The definition of the syntax is shown in Figure 8. The *radiation therapy* portion of system, *therapy*, can now be rewritten to provide a choice of type of *radiation therapy* depending on the patient's needs. This change is made using the new defined structure, *radiation therapy*, twice (see Figure 9).

$$\begin{pmatrix}\text{treated}\\ \text{patient}\end{pmatrix} = \begin{matrix}\text{radiation}\\ \text{treatment}\end{matrix}\begin{pmatrix}\text{patient,}\\ \text{machine,}\\ \text{personnel,}\\ \text{machine OK}\end{pmatrix}$$

$$\Big|\ \text{radiation therapy}$$

$$\begin{pmatrix}\text{patient1,}\\ \text{machine1,}\\ \text{personnel1,}\\ \text{OK}\end{pmatrix} = \begin{matrix}\text{run}\\ \text{machine in}\\ \text{chosen mode}\end{matrix}\begin{pmatrix}\text{technician,}\\ \text{person on table,}\\ \text{machine}\end{pmatrix}$$

Figure 8. Syntax.

System development efforts where systems possess many common structured patterns can benefit significantly from the use of structure abstractions. The node savings when using a defined structure is:

$$\frac{u \times n}{u \times (p+1) + s} = e$$

Where u = number of uses,
n = nodes specified wthout use of structures,
p = "plug-in" nodes specified for structure use
s = structure size (number of nodes on structure)
and e = efficiency

In the case, for example, where a definition has 40 nodes and one plug-in function, where each use is 1 plug-in node plus the parent node (i.e., 2 nodes) and it is used 1,000 times, there will be 2,040 nodes with the use of defined structures instead of 40,000 nodes (i.e., almost 20 times more efficient with defined structures, with a savings of 37,960 nodes).

As with an FMap, defined structures, in the form of parameterized types, can also be used to create a TMap. A *parameterized type* is a data type where the set of values is a set of types having a common data structure. The use of a defined structure on an FMap is analogous to the use of a parameterized type on a TMap. As an example, part of the TMap for system *therapy* is shown in Figure 10. Here in TMap,*therapy lab*, type, *patient*, is

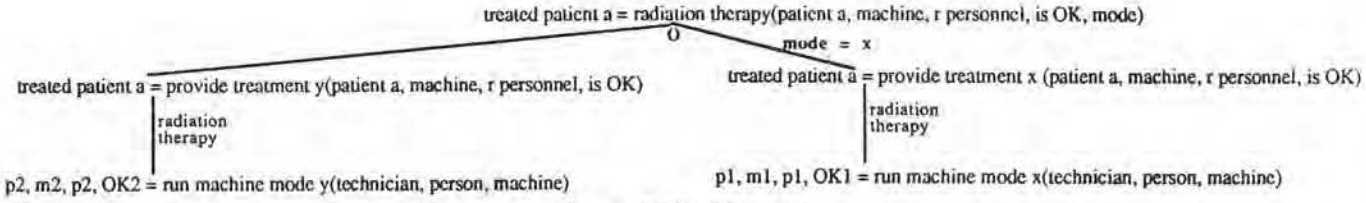


Figure 9. Use of Structure.

decomposed in terms of parameterized type,*TupleOf*, into its offspring types, *name* , *person*, *treatments* and *prescriptions*; *treatments* in terms of parameterized type, *OSetOf*, and *treatment* in terms of parameterized type, *OneOf*. *TupleOf* is a parameterized type used to define an abstract type that is a collection of a fixed number of different types of objects; *OSetOf* is a parameterized type that is used to define an abstract type that is a collection (in a linear order) of a variable number of the same type of objects; and *OneOf* is a parameterized type that is used to define an abstract type that is a set of abstract object types of different types from which one type is

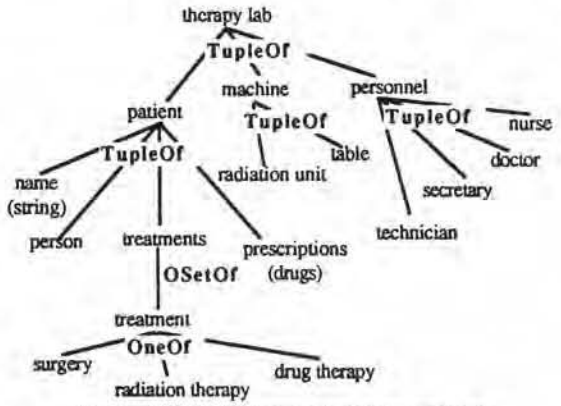


Figure 10. A Type System for Therapy Objects.

selected. A *parameterized type* represents a definition of a common pattern for types with the same structured relations. A parameterized type, a more abstract data type structure than a primitive one, provides the same generic primitive operations for any type decomposed with it. Whereas the same type of operations can be applied to types decomposed with a particular parameterized type, the same type of objects can be applied to functions decomposed with a particular defined structure. An instantiation of a parameterized type is a structured type (Figure 11). A *structured type* represents a definition of a common pattern for types with the same structured

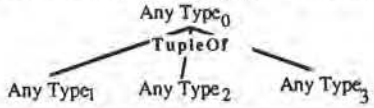


Figure 11. Syntax for Parameterized Type TupleOf.

relations. For example, an instantiation of OSetOf is *patient treatments*. *Patient treatments* is therefore a structured abstract type. An instantiation of a structured type is a structured object. A *structured object* represents a definition of a common pattern for states of an object with the same structured relations. Thus, a particular *patient treatment* is a structured object.

Types are defined indirectly on a TMap with primitive structures or parameterized types or defined directly as algebraic types with a set of operations and a set of axioms which define the relationships between these operations (see, for example, Figure 12).

The TMap tree topology of parameterized types provides a basis for the characterization of the control of objects in terms of their spatial relationships. An object of a parent type conceptually represents the containment of the objects of each of its children types which are component parts. (A type may be a non primitive type, a primitive type, a reference to an object type outside of its parent domain, an object type which is defined elsewhere as part of its parent's domain

```
data type: drugs(of t);
primitive operations:
                t    = top(drugs1);
          drugs2     = remaining (drugs2);
          boolean    = d equals(drugs1,drugs2);
          drugs2     = combine(t,drugs1);
axioms:
   where t is a T;
         d, d1, d2 are drugs (of T);
         reject is a constant drugs (ofT);
         empty is a constant drugs (of T);

   top(empty) = reject;
   remaining(empty) = reject;
   top(combine(t,d)) = t;
   d = combine(top(d), remaining(d));
   d equals(d1,d2) = equals(top(d1), top(d2))
         and d equals(remaining(d1), remaining(d2));
end drugs(of T);
```

Figure 12. Data Type drugs (of T).

or a recursive Type.) Each parent on a TMap and its children are used as parameters to a parameterized type that decomposes that parent into its children. A parent type replaces the "type" parameter in the parameterized type operation and each child replaces the "child" parameter in the parameterized type operation. This resolution results in an abstract type (Figure 13).

Each parameterized type has a set of primitive operations associated with it for its use [19]. As a result, all types decomposed with the same parameterized type inherit the same primitive operations. *Move to* is an example of a primitive operation associated with all parameterized types (Figure 13). It is therefore a

```
type: p1
      type = k:type(TMap)
      ...
      child = move to:child:type(type)
type: p2
      ...
```

Figure 13. Parameterized Types.

universal primitive operation. The universal primitive operations are used for controlling objects and object states. The FMap, Check, in Figure 14, uses the *MoveTo* operation with abstract type, *treatment* (Figure 15), to access the type of *treatment* and check if the *treatment* is to be *radiation therapy* or not. The universal primitive operations

include the ability to create, destroy, copy, reference, move, access a

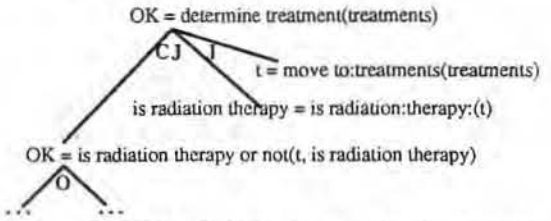


Figure 14. FMap which Uses Automatically Generated TMap Primitives for Types Treatment and Radiation Therapy.

```
type: treatment
        treatment = k:radiation therapy:treatment(TMap)
          boolean = is radiation therapy:treatment(treatment)
radiation therapy = move to:radiation therapy:treatment(treatment)
                    . . .
```

Figure 15. Excerpts of Abstract Data Type Treatment Automatically Generated from TMap.

value, detect and recover from errors and access the type of an object. They provide an easy way to manipulate and think about different types of objects. With the universal primitive operations, building systems can be accomplished in a more uniform manner. The universal operations also provide a semantic base that can be used to analyze the behavior of their interaction. (Constraints can be placed on the allowed interactions. These constraints can then be used by a constraint analyzer to eliminate a subtle class of user intent errors.) For example, the abstract type, *treatment*, in Figure 10, inherits the behavior of its TupleOf parameterized type. A treatment table, type, *table*, is the type of object that has a fixed number of components of different types. It has two components: a top and a fixed number of legs. The legs are defined with parameterized type, ArrayOf. The top may be removed from the table with the *get:top:table* primitive; or, a top can be put onto the table with the *put:top:table* primitive operation. These operations do not change the conceptual shape of the object. A table object without a top is still thought of as a table object; the fact that a table does not have a top does not change the fact that if it is a table object, it *could* have a top.

Definition of Objects with TMap

Whereas an FMap contains knowledge about the timing of objects, a TMap contains knowledge about the spatial organization of objects. TMap is used to create, manipulate, and understand the behavior of objects used in an FMap.

An object can exist in an active or a passive state. An object as a member of a data type is passive. That same object, as a function, is active. An object can "be" and "do" at the same time or interchange these respective roles. A medical data base in its passive state would be received as input and/or produced as output of a function; e.g.,

new medical database = organize(medical database)

But if the medical data base takes in an object as input or produces an object as an output it is in its active state; e.g.,

$$\text{medical database}_{\text{index}} = \text{medical database(index)}$$

In the example below, the *medical database*, as a function, is viewed as an active object with particular operators, as inputs, applied as passive objects. Here, the relationship between the *medical database* and its offspring, *people* and *materials*, is an *Include* (see Figure 16). This example, which conforms to the fact that type, *people*, is independent of type, *materials*, has an allowable set of interfaces for *medical database* as a function.

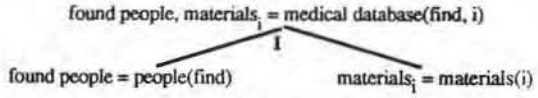


Figure 16. Independent Types.

Similarly *materials*, as an active object, is shown below, in Figure 17; where a decision is made, with an *Or* structure, to partition the set of *materials* into *drugs* or *radiation machine* materials.

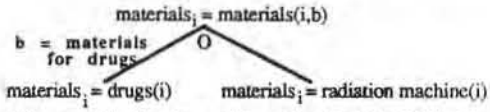


Figure 17. Partitioned Types (for Decision Making).

And, *people*, as an active object, is shown below, in Figure 18, where *people found* is created by the dependency of the object, *all the other people*, on object, *person*. In this case the relationships of *people* and its offspring is controlled with a *Join* structure.

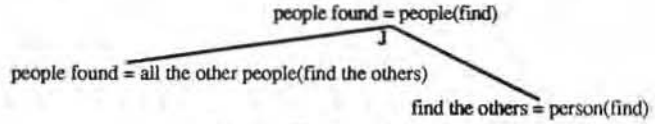


Figure 18. Dependent Types.

If, however,

new medical database = reorganize(medical database(index));

then medical database is doing with respect to index but being with respect to reorganize. An example is when a machine or a person as a function, *A*, is interrupted during its execution and is scheduled on a queue as a process to continue later; here, object, *A*, transitions from an active to a passive state. Likewise, *A* transitions from a passive state to an active state once *A* is removed from the process queue and begins once again to execute. (See, for example, Figure 19 where *A* and *B* are scheduled as passive objects and executed asynchronously as active objects.) Structures with asynchronous processes are discussed further in a section below.

An *object* is defined in terms of an *Object Map* (OMap™) by an operation in an FMap system. An object is a member, or instance, of a type. When a complex object is created, placeholders for all of its component objects are created (e.g., if a patient, Fred, is created, placeholders for his arms and legs are created). A *state* of an object is defined in terms of a *State Map* (SMap™) from an OMap system by an FMap operation. A state is an instance of an object. An *Execution Map* (EMap™) system is an FMap system with a comple instantiation of all objects plugged in for one performance pass of the entire system. It is the result of an execution of an FMap. An FMap shows all the possible lines of control while the EMap shows the actual lines of control taken for a particular execution phase. Recursion is made explicit at the EMap level. A recursive definition controls the tree extension process of an EMap by placing a "test" function (see, for example, *machine OK* in Figure 2a) to determine when and how to stop the extensions. A series of repeated functions that are not nested can be converted to a nested or recursively defined system by providing stopping conditions for the extensions. Functions that exist in different recursive extensions of the map can be performed in parallel if the functions are independent.

The patient, Fred, in terms of the aforementioned system viewpoints, could be a type, *person*. (See type, *person* in the TMap for the *medical database* (Figure 10). Whereas a TMap would contain type *person*, an OMap would have a specific person decomposed, e.g., Fred with his own arms, legs and head. An SMap would define a particular state of Fred (e.g., one SMap would define Fred with a broken leg at one time, another SMap would define Fred with a healed leg at another time). An FMap could refer to Fred as an input object or an output object in a system (e.g., an X-ray machine process could accept Fred both as an input object and return Fred as an output object.) An EMap for this FMap might show Fred at one time with a broken leg and Fred at another time with a healed leg, all in one performance pass along with other objects whose states have been instantiated.

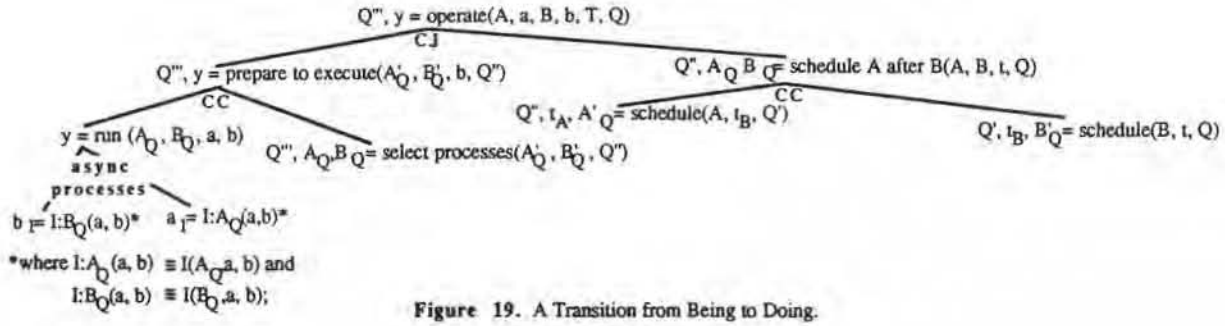


Figure 19. A Transition from Being to Doing.

In essence (see Figure 20), a State runs on an object "machine", an object on a type "machine", and a type on a parameterized type "machine" where

behavior (x) = PType((Type(Object(State(x)))))

and where, for example, x is "table" where the state is a new treatment table, the object is a treatment table, the type is a table and the parameterized type is a TupleOf.

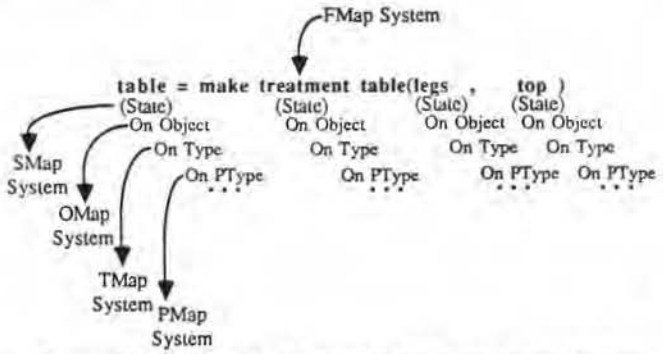


Figure 20. An Example of "Behind the Scenes" 001 System Mappings.

Given these various hierarchical definitional states or forms of control maps, the means is provided to define parameterized type mapping, type mapping, object mapping, state mapping, execution mapping and the integration of these mappings. Functions are under control with an FMap. Objects are under control with a TMap. A complete system (i.e., the integration of FMaps and TMaps) is under control since the input and output objects (and their states) of an FMap are defined in terms of a TMap.

The TMap properties ensure the proper use of a TMap by an FMap. A TMap has a corresponding set of control properties for controlling spatial relationships between objects. One cannot, for example, put an object into a data structure where an object already exists; conversely, one cannot remove an object from a structure where there is no object; a reference to the state of an object cannot be modified if there are other references to that state in the future and reject values exist in all types, forcing the FMap user to recover from them if they are encountered. A more detailed description of the TMap theory and its capabilities can be found in [11, 12, 13].

The type definitions discussed above were used to define application objects (e.g., a patient). Sometimes the application object could be the definition of an application type system, itself. For this kind of object a meta type is used. Data type, TMap, is a meta type whose operations operate on TMap definitions of an application. A TMap abstract type is defined with the parameterized type, *TreeOf*. TreeOf has a variable number of the same kind of components whose ordering is hierarchical. The operations of the type, meta type, can be used to access information about the types of a particular application. This information might in turn be used by a system that is knowledgeable of a general class of application systems. For example, an expert application system could search the TMap of a particular application (e.g., the patient TMap defined earlier) and determine its relationship to classes of patient applications. This might be, for example, accomplished by an expert system by determining if a certain set of types existed in the application type system. One of the primitive operations associated with type, TMap, can be used in this process to access the name of the abstract type at a TMap type node. The operations of another meta type are used to manipulate maps of objects regardless of their map types or application. This kind of meta type, OMap, has object maps as values. The primitive operations for type, TMap, and type, OMap, can be found in [19].

Data type, TMap, operations allow one to understand an object in terms of its type and the relationships of its type to other types in the system. Data type, OMap, operations allow one to treat all objects in the same manner regardless of their type. An abstract type allows one to distinguish one class of object from another class of object.

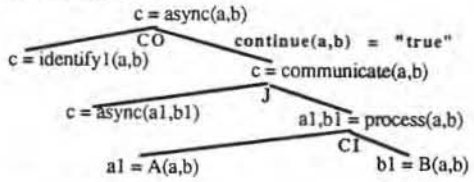
Sometimes it is desirable to layer one type onto another type (e.g., patient onto human in one architecture or the same patient onto array in another architecture; or a coordinate system for radiation onto a matrix system followed by the matrix system onto a vector system. The layering capability of 001 provides a means for secure, independent and open systems development within each object's environment. This layering capability is important on a large project, especially when different organizations are involved in the development of a system.

Real-Time Asynchronous Communicating Distributed Systems

Abstract structures can be defined for particular types of behavior in a system. An example of such a structure is a real-time, communicating, distributed, asynchronous structure. Async (where the Async system in Figure 21a is a variation) is an example of such a structure [20]. Async was defined in terms of the primitive structure, Join (J), and the non-primitive structures, Coor (CO) and Coinclude (CI), both of which were defined in terms of the primitive structures. The syntax part of the definition for Async is shown in Figure 21b. In Async the function parameters A and B are instantiated with functions which are consistent with the input and output types associated with functions A and B. The lines of control taken during the execution phase of Async are illustrated by an EMap (Figure 21c) which identifies how the network of primitive functions are activated over time and resources. The flow of objects between the activated functions can be thought of as an acyclic graph where nodes on the graph correspond to functions and arrows on the graph correspond to objects (which go from a source function to a target function). The Async structure defines the recursive interaction between multiple invocations of functions *A* and *B* (see Figure 21c). If there are two performer resources P1 and P2 that are able to execute the functions *A* and *B* (i.e., P1:A and P2:B), then we can use Async to control their functional intercommunication with respect to A and B. The

where a, a1, b, b1, c are requirements

a. Map



b. Syntax

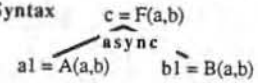


Figure 21a and b. Structure: Async.

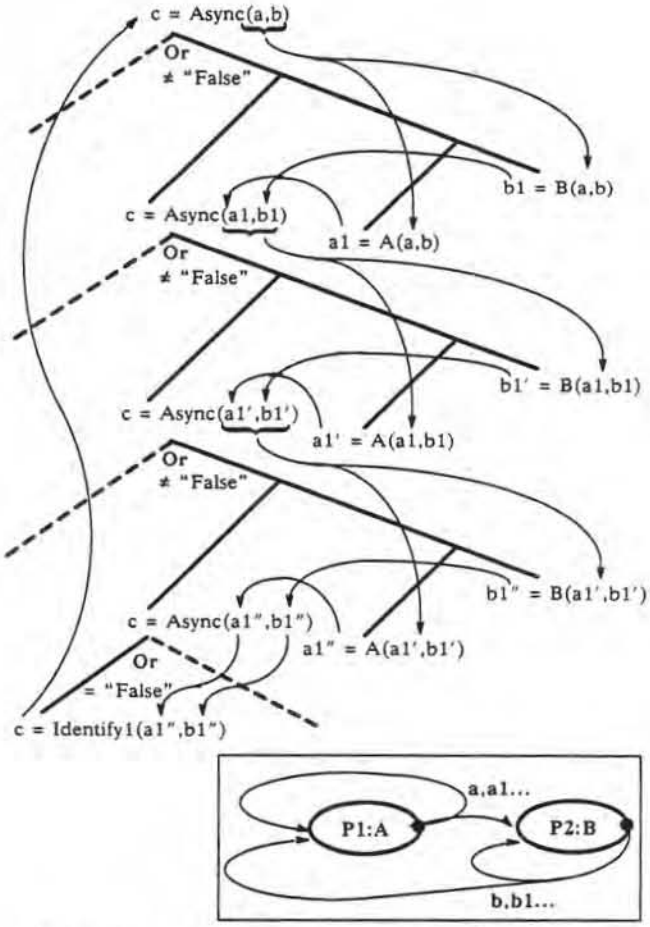


Figure 21c. An Execution of Async and the Communication Paths Between A and B.

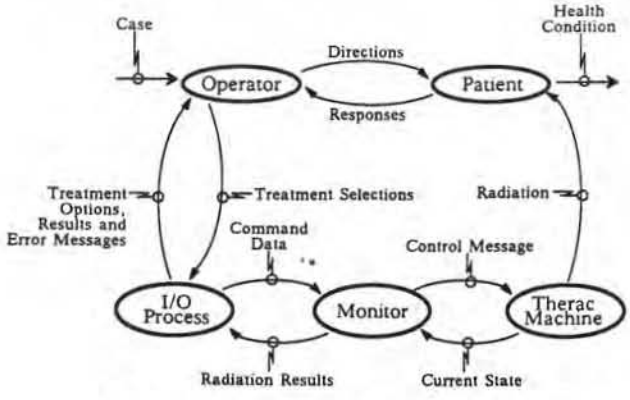


Figure 22a. A Radiation Therapy Environment.

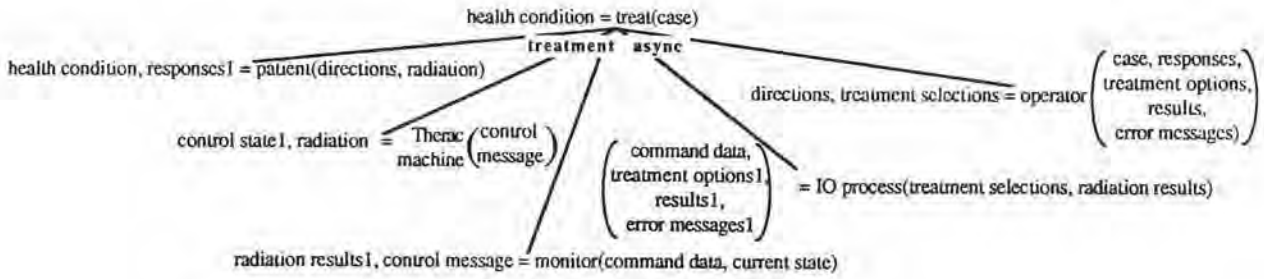


Figure 22b. A Radiation Therapy System.

EMap shows the successive activations of *A* and *B* on P1 and P2 and the flow of information or objects between them (a, a1, a1' . . . and b, b1, b1').

The radiation therapy environment in Figure 22a is an example of an asynchronous, real-time, communicating, distributed system. Figure 22b is one model from this environment which makes use of a multiple Async structure. Here, all of the offspring of parent, *treat*, are processes whose relationships with respect to each other are those as defined by the Async structure. Several real-time environments have been modeled using Async, Interrupt, Update Interrupt, Communicate and other real-time structures [14, 21, 22, 23, 24].

The Automation of the Technology

The automation of the technology is an integrated tool suite for automatically developing ultra reliable models, ultra reliable simulations and ultra reliable software systems (see Figure 23). It is based on a philosophy that a reliable system is developed in terms of reliable systems. The tool suite supplies the building blocks to build a reliable system; as each new system is created by the user, it, in turn, can be used as a building block.

A model captures all known properties of a system at a given time. It consists of an integration of functions and data types for a target system (for example, a real-time, asynchronous, communicating, distributed system) which resides on a combination of hardware, software and humanware environments. Once a model is defined, complete source code can be generated for that model. Rapid prototyping with "graceful evolution" or production can then proceed; the model is simulated to observe its behavior in various dynamic states; software portions of the model become fully implemented system(s).

The tool suite embodies a hierarchical structured network modeling approach based upon a formal concept of control. A model is decomposed using a control map which ensures that the relationships of all objects, in all states, are under control. Its definition language, 001 AXES, is both functional and object oriented. Functions are decomposed with an FMap control map. Data types are decomposed with a TMap control map. The FMap is used to connect functions with objects as input and output which abide by the rules of the data types in its associated TMap.

At any level of construction, a model may be submitted to the Analyzer component of the tool suite. The Analyzer ensures that both data and functions are used in a consistent and logically complete manner, eliminating approximately 75% of all system errors.

Once a model is decomposed to the level of existing libraries and it has been successfully analyzed, it may be handed to the Resource Allocation Tool (RAT) component of the tool suite, which will generate logic flow from both the function and the type control maps. This logic is automatically connected to previously existing function and type primitives in the core library, as well as, if desired, libraries developed from earlier implementations. The generated source code can be compiled and executed on the same machine on which the tool suite resides or it can be ported to other machines for subsequent compilation and execution.

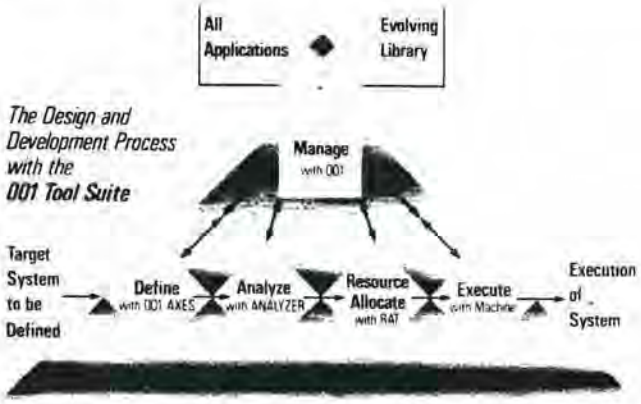


Figure 23.

Because of its features of reliability, automation, abstraction and reusability, the tool suite maximizes productivity. Systems can be designed, developed and maintained with minimum man-months in minimum time.

The tool suite contains the following components: an editor for supporting a user in defining his system graphically or in textual form; an executable specification language for defining logically accurate models that are consistent and logically complete for both functional and object oriented hierarchies and their integration; an Analyzer for automatically detecting errors according to a formal set of rules; an abstract type generator which generates a system of data types for a particular application domain from an object type hierarchy that is decomposed in terms of parameterized types; a multi-language source code generator, or Resource Allocation Tool (RAT), which produces code of the user's chosen form (e.g., C); and a documentor which produces a system definition, its implementation and its description.

With the tool suite, a system goes through a complete development process. Figures 2a, 2b, 7, 8, 9 and 22b show FMaps as they are entered into the automated system. Figure 10 shows TMap, *therapy lab*, as it is entered into the 001 system. Figure 24a shows the results of an automatic generation of primitives from this TMap and the source code (Figure 24c) which has been generated by the RAT, for the FMap, *therapy*, in Figure 2a. FMap, *therapy*, uses objects derived from TMap, *therapy lab*. Figure 24b shows the data type implementation file, produced by the automatic generation from the TMap, which was used by the RAT to automatically produce C source code (Figure 24c).

```
A. Generated Abstract Type Patient
   Data Type: Patient;

   Primitive Operations:
   Drugs,Patient=Get_Prescriptions_Patient(Patient)
         Patient=Put_Prescriptions_Patient(Drugs,Patient)
           ......

B. C Data Type Implementation File Used By RAT
           ......
   .TYPE:
   #include "hti_1:[hackler.demos.therapy.MEDDB]PATIENT.H"
   .DECLARE:
           DECLARE_PATIENT(%PRO1)
   .GET_PRESCRIPTIONS_PATIENT:
           GET_PRESCRIPTIONS_PATIENT(%PRO1,%PRO2,%PRO3)
           ......

C. C Source Code Generated By the RAT Using CDT Files
    /*THERAPY   CONTROL MAP: THERAPY translated to FUNCTION THERAPY
           by HTI-001 version 32.5/33.0/C-001-A
           at 12:47:31.54 hours on day 15-MAR-1988
           OR's BRANCH RIGHT ON "FALSE" BOOLEAN.
   */
   #module THERAPY
   #include "hti_1:[hackler.demos.therapy.MEDDB]PATIENT.H"
           ......
   THERAPY(V_PATIENT_A,V_MACHINE,V_R_PERSONNEL,V_MACHINEOK,V_PATIENT_B,
           V_DRUGS,V_D_PERSONNEL,V_TREATED_PATIENT_A,V_TREATED_PATIENT_B)
           ODECLARE_PATIENT(V_TREATED_PATIENT_A)
   {
   /*                  _____LOCAL VARIABLE DECLARATIONS_____
   */
           DECLARE_PATIENT(V_P)
           ......
   /*                           _____SOURCE CODE____
   */
           DRUG_THERAPY(V_PATIENT_B,V_DRUGS,V_D_PERSONNEL,V_TREATED_PATIENT_B);
   /*RECURSION ======> RADIATION_THERAPY <======*/
           ......
           for(L101=0; L101==0;)    /* Loop Head */
              {L101=1;
       if (V_MACHINEOK < 1)
           {  /* ----- BEGIN FALSE AND REJECT BRANCH ----- */
           WAIT(V_PATIENT_A,V_MACHINE,V_R_PERSONNEL,&V_P,&V_M,&V_R,&V_OK);
           ......
           L101=0;                  /* Loop End */
           }
       else
           {  /* ----- BEGIN TRUE BRANCH ----- */
           TREATMENTS(V_PATIENT_A,V_MACHINE,V_R_PERSONNEL,V_TREATED_PATIENT_A);
           } /*END IF*/
           }; /* Restore Initial Value */
           ......
   return;
   }
```

Figure 24. Output Automatically Generated by 001.

The technology upon which the tool suite is based teaches or helps a person to think in a new and organized way about systems; it helps someone define his thoughts as simply as possible, but not simpler; it makes sure that a system definition is unambiguous before a system is implemented and it provides a means to build systems that are logically correct.

Project Management Considerations

The ability to build logically correct systems is important on its own right. There is, however, a direct relationship between having the ability to build logically correct systems and productivity. In recent productivity studies with the tool suite, where each system was developed for the first time within a given application environment, systems were produced with a productivity in man months of a range of 10:1 to 20:1 and a productivity in calendar time with a range of 3.5:1 to 7.5:1 [23, 25]. The productivity ranges are higher than this when a system is developed within the same application development environment as one that was previously developed with the tool suite. The reason is that the developers are more familiar with either the application, or the technology or both, and higher level libraries are used that were created on a previous application using the same technology.

Contrary to what one may expect, one need not totally reorganize either the corporate management structure or the life cycle model to use this kind of technology. It can be applied in an evolutionary manner. For example, in a typical life cycle model (see Figure 25) the 001 Axes and Analyzer components can be used to develop requirements and specifications. The difference is that the requirements and specifications will be more formally defined than with a conventional approach and there will not be interface errors. The design phase is one of choosing which RAT and which environment to "RAT to". (If a desired RAT does not exist, one is created.) There may also be a design process, in this phase, for some additional requirements for libraries. The implementation phase is one of "pushing the button" and "Ratting". Integration is inherently being performed throughout the life cycle process. There is still testing to be performed. There will be, however, approximately 75% less errors to search for at the start of the testing phase than in a traditional life cycle. And, certain kinds of tests are no longer an

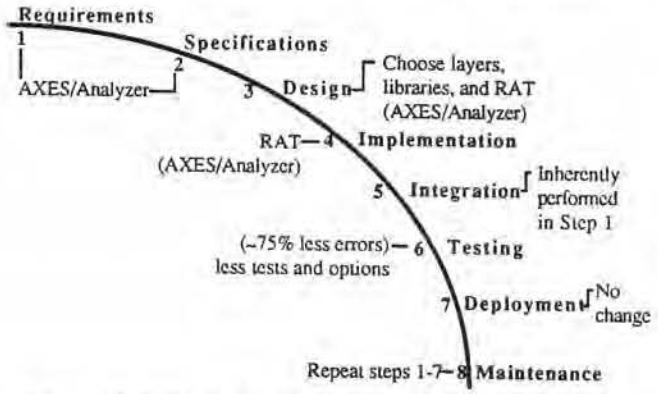


Figure 25. Evolving from the Conventional Model to the 001 Model.

issue (e.g., wire tracing). Maintenance is a repeat of development. That is, all changes to the system are made to the requirements/ specifications model and code will once again be automatically produced. Since maintenance traditionally accounts for 70-90% of a typical large development effort, [26, 27], a method such as 001 significantly impacts the costs normally attributed to this phase. Table 1 summarizes the overall differences of using a technology like 001 along with its automated tool suite as compared to building a system with traditional techniques.

Therac 25

Just as with avionic systems, missile systems and certain manufacturing systems, ultra-reliable systems are needed for the medical environment. The Therac 25 incident occurred because the systems involved within the Therac 25 environment violated many of the principles of a zero-defect system environment. (It, in fact, is not known not to be true for any other typical system and its

BEFORE	AFTER
Errors • 73% interface* • Most found **after** implementation • Some found manually (44%*) • Some found by **dynamic** runs • Some never found	**Errors** • No interface errors • All found **before** implementation • All found by **automatic** and **static** analysis • Always found
Inconsistent Requirements	Consistent Requirements
Documentation and programming are manual	Documentation and programming are automatic
No guarantee of function integrity after implementation	Guarantee of function integrity after implementation
Understandability, portability, repeatability not prerequisites	Understandability, portability, repeatability are prerequisites
Flexibility and handling the unpredictable not prerequisites	Flexibility and handling the unpredictable are prerequisites
Productivity • Not cost effective • Difficult to meet schedules	**Productivity** • 10 to 1, 20 to 1, . . . • Minimum time to complete 3 to 1, 4 to 1, . . .

*Source: APOLLO statistics

Table 1. A Comparison.

development environment.) The design of the Therac 25 system was overly complicated; there were system design problems in the integration of hardware, software and humanware functions; there was no back-up in the case of primary system malfunctions; there was a lack of hardware error detection and recovery mechanisms for its own malfunction; there were real-time software logic and timing errors; there was no back-up in the software for its own potential errors; poor communication existed between the subsystems (e.g., between the operator and the operator's maual); there was a lack of a formal definition of the system and a lack of a formal QA process. Any *one* of these violations either did result, could have resulted or still could result in a serious accident. We can learn from these incidents. There are steps that can be taken now in building future medical systems. They are summarized in Table 2.

Checklist for Building a System
•Formally model integrated system of hardware, software and peopleware with experts involved from all disciplines and with automated assistance. - define a TMap for all system objects - define an FMap for all functions the system should perform - define all errors that could happen - prioritize errors - incorporate protection from errors in system design - incorporate back-up protection from catastrophic errors in system design - build abstractions and reiterate the process • Automatically analyze model for ambiguities (i.e., inconsistencies and incompleteness of logic). (This step will prevent logic and timing errors in system design and in software produced from it.) • Automatically produce code for simulating model and/ or for developing software • Simulate and test • Go back and change model until it is satisfactory to all system experts • Build rest of system to go with software • Perform QA process with independent parties and with members from software, hardware and humanware in each case where it is applicable.

Table 2.

Summary

The ability to develop systems with techniques approaching zero defects will not just happen by acquiring a technology and its associated tool set. What is needed is a mind set to define systems formally and a determination to relate to and adhere to the set of processes of modeling, simulation and software development as a scientific method. Once the decision has been made to follow the formal path, training proceeds.

Standards are established and the technology is applied. The technology can be applied in either its manual or automated form. Significant benefits, in fact, have been obtained by engineers and scientists who have used these kind of techniques manually. The automation of the technology helps to ensure that the technology is applied correctly.

Although the transition to incorporate these techniques requires an adjustment, the benefits to the medical community received from approaching zero-defect systems are far reaching. In some medical environments, systems will be possible to build that were not affordable in the past; in other medical environments, many tragic incidents, such as those which not uncommonly and not infrequently occur in all of today's most respected medical centers, will be eliminated.

This paper was presented at the IEEE Symposium on Policy Issues in Information and Communication Technologies in Medical Applications, Rockville, Maryland, September 29, 1987.

Acknowledgement

The author would like to express appreciation to Ron Hackler for his contributions and for a most helpful and critical review of this paper; the author would like to thank Nancy Krohngold for technical editing assistance, electronic publishing assistance and for preparation of this manuscript.

References

1. Atomic Energy of Canada, Ltd., Medical Products Division, "Corrective Action Plan -- Tyler and Yakima Incidents," July 17, 1987, and letters dated 5/5/86 and 3/2/87 from Walter Downs, Manager, Quality Assurance, Atomic Energy of Canada to Ed Miller, Device Monitoring Branch, Center for Devices and Radiological Health, Food and Drug Administration.

2. E. Joyce, "Software Bugs: A Matter of Life and Liability," *Datamation*, vol. 33, no. 10, pp. 88-92, May 15, 1987.

3. M.A. Farber and Lawrence K. Altman, "A Great Hospital in Crisis," *The New York Times Magazine*, January 24, 1988, pp. 18-21 ff.

4. William Morris et. al., *The American Heritage Dictionary Second College Edition.* Boston: Houghton Mifflin Company, 1976, p. 806.

5. M. Hamilton and S. Zeldin, "Higher Order Software -- A Methodology for Defining Software," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, March 1976.

6. M. Hamilton and S. Zeldin, "The Relationship Between Design and Verification," *The Journal of Systems and Software*, vol.1, no. 1, pp. 20-56, 1979.

7. M. Hamilton, "Zero-defect Software: the Elusive Goal," *IEEE Spectrum*, vol. 23, no. 3, pp. 48-53, March, 1986.

8. A. Razdow and R. Hackler, private discussions relating to the implementation of the predecessor tools to 001™, Cambridge, Massachusetts, 1980-1985.

9. J. Martin, *System Design from Provably Correct Constructs.* Englewood Cliffs: Prentice-Hall, Inc., 1985, pp. 3-379 passim.

10. M. Hamilton and R. Hackler, evolving lecture notes, Hamilton Technologies, Inc., February, 1986 - March,1988.

11. R. Hackler, "Structured Relations Between Objects," in *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, Honolulu, HI*, January, 1984.

12. R. Hackler, "TMap Basics" (Support Memo No. 3). Cambridge: Hamilton Technologies, Inc., December, 1987.

13. M. Hamilton, lecture notes on Foundations of 001 Technology, October, 1987.

14. M. Hamilton, "Design Properties and Their Impact on Automation," Cambridge: Hamilton Technologies, Inc., T.R. No. 2, January, 1986.

15. R. Hackler, Prototype version of 001 tool suite, Hamilton Technologies, Inc., 1986.

16. R. Hackler, R. Poirier and J. Feldman, Versions 1.0 and 2.0 of 001 tool suite, Hamilton Technologies, Inc., 1987.

17. D. Weinreb and D. Moon, "Flavors: Message Passing in the Lisp Machine," Cambridge: Massachussetts Institute of Technology Artificial Intelligence Laboratory, A.I. Memo No. 602, November, 1980.

18. M. Hamilton, "Design of the GN&C Flight Software Specification," Cambridge: Charles Stark Draper Lab., Doc. C-3899, Feb., 1973.

19. Hamilton Technologies, Inc., *The 001™ System Reference Manual.* Cambridge: Hamilton Technologies, Inc., 1987, Appendices A-D.

20. M. Hamilton, "The Ada Environment as a System," *Proceedings of The Ada Environment Workshop*, sponsored by DoD High Order Language Working Group, Harbor Island, San Diego, CA, November, 1979.

21. M. Hamilton and R. Hackler, "Oil Discovery Example," Cambridge: Hamilton Technologies, Inc., T.R. No. 1, January, 1986.

22. R. Hackler, "Real-Time and Distributed Modeling," (Support Memo No. 4), Cambridge: Hamilton Technologies, Inc., December, 1987.

23. Hamilton Technologies, Inc., "Interim Report to Rexham Aerospace and Defense Group, Task 2, Subtask 1," Cambridge: Hamilton Technologies, Inc., March 7, 1987.

24. University of California Los Alamos National Laboratory Contract No. 4-X28-8699F-1: Defensive Technology Evaluation Code (DETEC) Conceptual Model Development, 1-14-88, Hamilton Technologies, Inc.

25. Hamilton Technologies, Inc., "Final Report to McDonnell Douglas Astronautics Co. (Huntington Beach): HOE Demo System," Cambridge: Hamilton Technologies, Inc., November 3, 1986.

26. J. Martin, *An Information Systems Manifesto*, Englewood Cliffs: Prentice-Hall, Inc., 1984, pp. 143-194.

27. D. Stamps, "CASE: Cranking Out Productivity," *Datamation*, vol. 33, no. 13, July 1, 1987.

About the Author

MARGARET H. HAMILTON is President of Hamilton Technologies, Inc. of Cambridge, Massachusetts, a company she founded in January, 1986, to develop methods for producing reliable models, reliable simulations and reliable software systems.

From 1976 to 1985 she was president and founder of Higher Order Software, Inc. From 1965 to 1976 she headed the software engineering division at the Charles Stark Draper Laboratory of Cambridge, during which time she managed the on-board flight software project for the Apollo space missions.

Hamilton, along with her staff, created a theory for designing and developing reliable systems. This theory, which began in the late 1960's, formed the beginnings of the 001 technology. The 001 technology, along with its associated automation has evolved, uninterrupted, under her leadership over the last twenty years.

001™: The Thinking Person's Product for Building Systems

Hamilton Technologies, Inc.

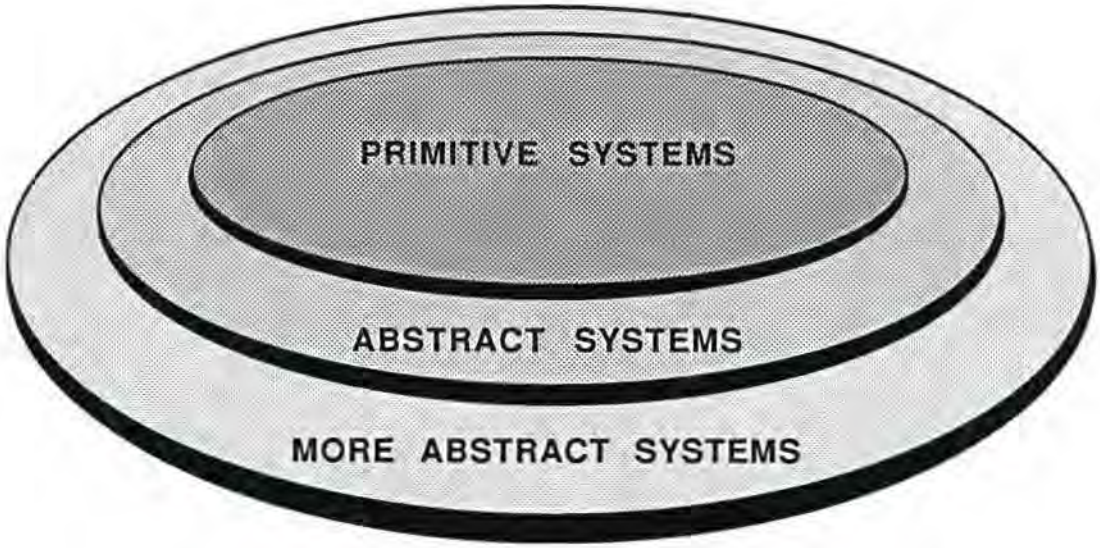
HTI

001™: The Thinking Person's Product

001™, a product of Hamilton Technologies, Inc. (HTI), is a new generation product for designing and producing software systems. It is a product that is all about organizing your thoughts in such a way that thinking can be a reliable and productive process; it is intended to teach or help a person to think in a new way about systems. 001 accomplishes this by helping someone define his thoughts as simply as possible, but not simpler*; it makes sure that a set of thoughts, or system definition, is unambiguous before a system is implemented. It provides a means to build systems that are logically correct. 001, a computer-aided reliable thinking product, is a solution towards developing reliable models, reliable simulations and reliable software systems.

Philosophy of 001

The philosophy of 001 is that reliable systems are defined in terms of reliable systems. This is how you make it work: Use only reliable systems; integrate these systems with reliable systems; the result is a system(s) which is reliable. Then, use the resulting reliable system(s) along with the more primitive ones to build new, more abstract and larger reliable systems.



001 provides a core set of reliable primitive data types, reliable primitive functions and reliable primitive mechanisms to connect functions to data, functions to functions and data to data. All 001 systems can be developed with these primitives or more abstract systems defined in terms of these primitives.

You Can Manage and Integrate the Life Cycle with 001

The concept of 001 began with APOLLO. At the time of APOLLO 11, Margaret Hamilton, now founder and president of HTI, was then Director of the APOLLO on-board flight software effort. Hamilton performed empirical studies on the APOLLO software environment with her staff in order that her people and others could benefit from this experience for future software and related efforts. The result was a mathematical theory which when used properly would allow one to define a system in such a way as to prevent "before the fact" interface errors, or errors of ambiguity.

---

*"Everything should be made as simple as possible, but not simpler." --Albert Einstein

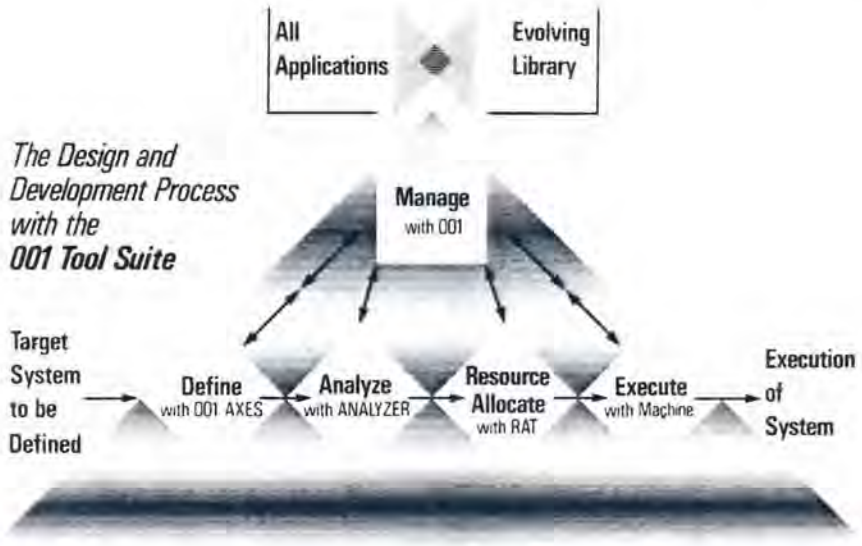
Once Hamilton and her staff realized that systems could be defined unambiguously, it was clear that system definitions (or models) could be analyzed for interface correctness. Furthermore, this finding led to the fact that source code could be automatically produced from models which are unambiguous. The staff of HTI has continued to build upon this theory as a foundation for the 001 product whose sole purpose is to help you develop reliable real world applications.

001 is an automated development product that spans and integrates the entire life cycle of application development. 001 is more than a user friendly front end. It also does the work.

The key is the mathematical systems theory behind 001. Because of its foundations 001 enables the user and developer to produce requirements specifications free of ambiguities. 001 will then automatically generate source code of the user's chosen form along with technical documentation from the reliable specifications. This source code exactly reflects the user's specifications throughout the development and maintenance process.

Not only does 001 deliver systems that are demonstrably reliable, regardless of their complexity and size, but it also delivers them more quickly than ever before.

001 is currently available on the DEC VAX under the VMS operating system.



001 Provides a Means for you to Define Logically Accurate Requirements Specifications

Traditional software development involves many manual and disjoint steps. The system requirements are often presented in a verbal form and/or produced in a written form by users, each in his own terms. Then they are manually translated into requirements specifications written after numerous iterations between the users and the systems analysts. This is both an error-prone and lengthy process.

001 offers a solution to this manual process.

With 001, using a computer, both the users and the system analysts can all participate in an integrated requirements session together.

With 001's interactive graphics component, system requirements can be easily captured and defined in machine readable form using the 001 AXES requirement specification language. The computer becomes the repository for both functional requirements and the requirements of all of the system objects. Both functional and data requirements can be shared across all areas of an organization. 001 assists you in decomposing functional and data requirements where high level system requirements are elaborated into more detailed requirements; it represents them in control maps that show and trace all possibilities of priority, ordering, logic and data flow throughout the system.

Hardcopy output of the requirements may be produced at any time during the development process. This output may be used in system reviews with management, end users and developers for greater clarity and communication in the system development process.

After specifying the requirements, the next logical step in software development is the design process. This process is also managed and integrated by 001. With the graphics component, the same control map technique is used to specify functions and data at the design stage as was used during the requirements specification stage of development.

The Use of the 001 Axes Language Controls your System and its Development Process

With the 001 AXES language all systems requirements are defined by decomposing a system using a control map.

The functions are decomposed using structures which connect them by 001 rules using a function map (FMap) control map. The data types are decomposed with structures which connect them using 001 rules using a type map (TMap) control map.

The FMap connects the decomposed functions with input and output data which abide by the rules of the data types using TMap. The result is an integrated set of reliable requirements which set the foundations for the management, implementation and operation of a complete system environment and its development.

001 Automatically Analyzes your Requirements and Design

Once the functions and the data for a system have been elaborated to the level of design which interfaces to existing function libraries and data libraries within the 001 environment, the system is submitted to 001's Analyzer component. The mathematically based Analyzer not only ensures that the data is used in a consistent and logically complete manner, but that the interfaces between the functions and between the data are consistent and logically complete. Studies have shown that approximately 75% of all system errors are errors of ambiguity (i.e., interface errors), which are errors of inconsistency and logical incompleteness. These include errors of timing, ordering and priority. These are all found by the Analyzer.

The Analyzer finds the interface errors early in the software life cycle, where they are the least costly to fix. Result: a consistent, logically guaranteed and integrated system design.

001 Automatically Provides you with Generated Code and Documentation

In the traditional development cycle, the hand-coding process from the written design specifications would be the next step. 001 virtually eliminates this step.

001 will automatically generate logic flow from both the functional and the data control maps and automatically connect this logic to previously existing function and data primitives in its core

library. The option exists for system implementors to build new libraries to hook into or use existing libraries from other implementations should it be desirable to do so.

The generated source code can be compiled and executed on the same machine on which 001 runs or it can be ported to other computers for subsequent compilation and execution.

Complete technical documentation is produced automatically from the code generation process, including the annotated source code. Additional comments can be inserted by the user into the requirements specifications using the graphics editor. These comments are also automatically generated along with the automatically generated technical documentation.

Both the source code and technical documentation is consistent with the system design and requirements specifications. Result: significantly reduced effort necessary to fully document and put your system into production.

001 Minimizes your Performance Testing Effort and Helps you to Rapid Prototype

With 001, the performance testing effort is significantly reduced since the generated code is free of ambiguities and it is consistent with both the requirements specification and software design. In fact, it is not necessary to test for the approximately 75% of the errors that would have still existed at this time in a traditionally developed system. The remaining tests that still exist at this time concentrate only on answering the question "Is this what the user really wants?"

Once the system (or a part of it) has been analyzed successfully and source code has been automatically produced from it, a design can be interactively demonstrated through 001's prototyping capability. During this phase both the user and the system designer can validate the system by executing different paths based upon interactive input. System components that are not complete at the time of prototype execution may be simulated.

At this point in the software development process, a model has been produced that is consistent and logically complete. And it can be demonstrated to the user by prototyping or simulation that his requirements indeed have been defined correctly by 001.

Once all of the parts of the system have been brought through this same process, they are completely integrated and the system is operational.

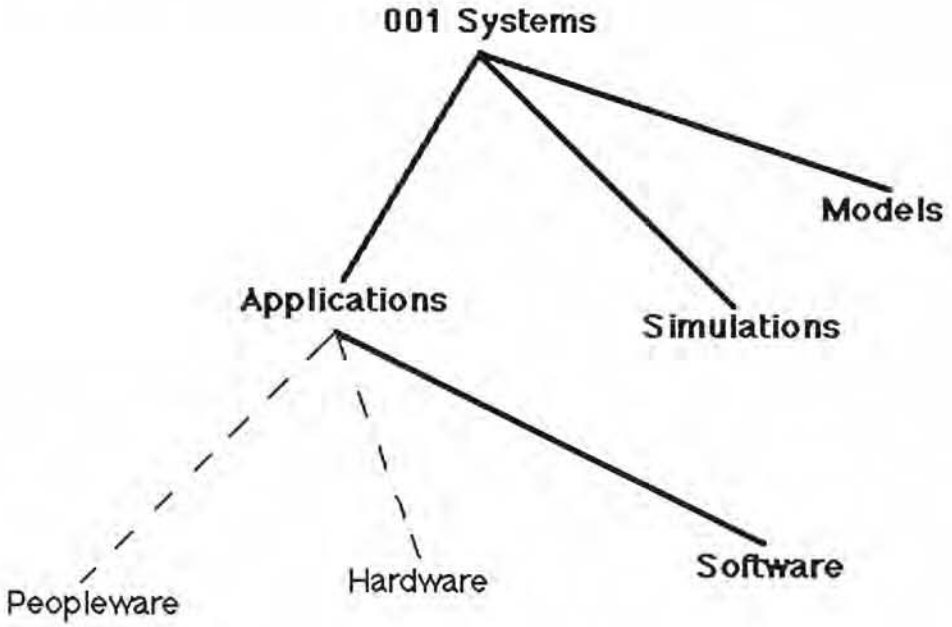
001 Will Give you the Same Benefits During Maintenance

001 treats the need for system modifications as an integral part of the development process, not as an afterthought. To make a change, you need only modify the control map that was used to define the system. The persons responsible for maintenance would use 001 in exactly the same way as a developer of the original system, including the Analyzer and RAT components and its prototyping capabilities. Revisions to the system are automatically accompanied by revised documentation and source code. No longer is it necessary for maintenance personnel to decipher the individual coding techniques of the initial system developers. 001 renders this process obsolete.

001 is a General Systems Building Tool

001 is a product for building systems according to a particular user's need. A model can be defined with 001 which captures all known properties of that system at a given time. Such a model could have a combination of functions and data making up hardware, software and peopleware. That same model can be simulated with 001 to show its behavior in various dynamic states. The software portions of that model can be developed with 001 from its requirements stage as a model throughout

its implementation and rapid prototyping stage as a simulation and finally as the actual target system that is to be developed.



With 001 you can Build Reliable Systems that you can Trust

Reliable Systems are Understandable

Traditionally systems have been hard to understand from the requirements phase throughout development and finally into the coding phase. There are several reasons for this. System definitions are ambiguous; users, developers, managers and computers at various levels of development more often than not speak different languages; the system is represented in a form whose focus is not integration; definitions are not friendly in that each modeler or programmer often speaks to himself in symbols only he understands; unnecessary detail is resident thought the development process; data cannot be traced within the system between phases and within a phase. And, often either too much detail exists or not enough detail exists to state a particular concept or algorithm.

001 directly addresses these issues.

001 forces systems to be unambiguous; it supports, users, developers, managers and computers to use the same language. That is, they have a common set of semantics with which to dialogue; yet 001 allows the syntax to be chosen by its users and thus the medium of communication is as friendly as desired; each data object path can always be traced throughout a 001 system. And integration is inherently a focus of the properties of 001; that is, as ambiguities are driven out, integration, a natural process of the 001 technique takes place.

Reliable Systems are Error-free

Error-free means no ambiguities. This means no inconsistencies, no redundancies, no logic that is incomplete. It means that the output of each development phase meets the requirements of the previous one.

001 directly addresses these issues.

The 001 AXES technique automatically prevents ambiguities from being allowed to stay in the development process.

To be error-free the system must perform the job the user intended it to perform.

001 addresses this issue.

001 limits the area of concentration for the verification of the user's intent by having had eliminated approximately 75% of the errors from the system with its mathematical rules. As a result what's left are only errors which fall into the category of user's intent. In addition, with 001's FMap and TMap capabilities in the requirements specification language, 001 AXES, the user need only define what it is he wants to do and the rest is automatically taken care of by 001 filling in the how in terms of 001's core primitives. With each project use of 001, and with each future enhancement of 001, HTI continues to narrow the gap of possible types of user intent errors remaining within a 001 user's environment.

Reliable Systems are Flexible and can Handle the Unpredictable

Flexibility and the ability to handle the unpredictable are put to the test during the operational phase of large, complex real-time systems. These systems must be able to detect errors and recover from them; they must be able to reconfigure to changing and unforeseen events. They must be able to handle asynchronous events by being asynchronous themselves. This means that they must be able to handle interrupts, asynchronous events and be able to distinguish between priorities.

Flexibility and the unpredictable are also put to the test when large, complex real-time systems are being developed or maintained. Changing requirements for these systems are often a major stumbling block for organizations and projects. As a result either requirements are not changed and the impact on the organization or project is an adverse one or the requirements are changed and the effects are error-prone and costly to the system. Again, an adverse effect on the organization.

001 addresses these issues.

Properties of 001 systems inherently lend themselves to change and the unpredictable.

The very fact that ambiguities do not exist and that each change is traceable eliminates many of the concerns that traditionally surface with the flexibility issues.

Single reference/single assignment is one of the properties of 001 systems. This property, in addition to giving you the facility to trace a variable thoughout the system, is the reason why the system is by its very nature modular and further, reconfigurable.

The main problem that traditional techniques have had with asynchronous systems is that these systems, because of their complex interfaces, are error-prone. 001 has rules which control, among other things, priority, ordering, data flow and timing so that asynchronous behavior is forced to be safe in terms of its interfaces.

The issue that remains is how to handle the concept of dealing with the unknown. 001 has core primitives which serve as a foundation for a real-time asynchronous and reconfigurable environment that is able to deal with events which are not expected and with errors in the system from which recovery can be made. The HTI staff is prepared to help you build systems of this complexity with 001.

Reliable Systems are Portable

Often when a system is developed there is a need to move it or parts of it to more than one application environment. The same algorithm could be used, for example in a missile system as is used in a communication system; there may be a desire for that same system to run on more than one computer or to be implemented in more than one language; that same system may run on one processor in one operational phase and several in another; there may be a need for that system to allow for a

plug-in replacement of different configurations of other systems within that same system. A layer of that system may be needed (e.g., the very top layer requirements) to work with various layers of other systems as lower layers; or there may be a desire for various layers in a particular system to be hidden from others either for security purposes or for separation of management in developing that system's layers.

For those systems which are not modular, portability of these various forms are not possible. The result is unnecessary redevelopment of the same systems.

001 directly addresses these issues.
001 will automatically produce by the very choice of a RAT by the user, an implementation to a chosen computer, language or implementation. The generated code will run in the chosen computer environment which has the compiler to go with the selected language. This feature alone allows for different application environments.

The 001TMap functionality automatically sets up the environment for layered and secure developments. Again, this feature lends itself to different application environments.

The 001 abstraction capabilities provide for a plug-in of different modules where a chosen structure, the basic mechanism for connecting functions, allows for a plug-in of functions chosen by the user but which at the same time provides the functions and their connections that are common to all uses of the same structure.

Reliable Systems Allow you to Capitalize on Repeatability

Software development organizations are notorious for re-inventing the wheel. Often, the same concepts, requirements, algorithms and code are created over and over again; the same mistakes are made with each new project. Lack of repeatability is the single largest reason for wasted time and dollars within an organization or a project.

001 directly addresses these issues.
Repeatability is the ability to do something over and over again. 001 capitalizes throughout the life cycle on automation, which is the ultimate form of repeatability. But existing forms of automation themselves can be automatically used over again with 001.

By providing a method to share primitive libraries as well as libraries developed in terms of these primitives, among various development projects, 001 allows the development organization to concentrate on the unique aspects of each system being developed while sharing libraries which are common across all systems and more libraries which are common across particular families of systems. The user of 001 has the capability of adding his own libraries to the 001 environment as well.

001's reusable library techniques add to significant productivity improvements. System models, along with their associated automatically generated code can be developed once, stored in reusable libraries and shared across multiple programs, projects and systems. This capability reduces the need to "reinvent the wheel" for every new system.

001, through its abstraction and distillation capabilities, both in functional decomposition and in data type decomposition, allows you to hide information whenever it is desirable to do so; yet should it be necessary to look at more detail within higher level abstractions, that information is readily and automatically available. All of the more abstract components of a system are forced to be defined in terms of more primitive components. Thus, the system is unambiguously defined and implemented throughout its implementation.

As a result of the powerful repeatability concepts of 001, any work that has been produced does not need to be wasted either within one development process or across many development processes.

Reliable Systems are Affordable:
Developed within Budget and Time Constraints

The marketplace is replete with stories about systems not being delivered on time or about overruns running in the millions. Sometimes a system never is delivered because there are no more dollars, time has run out or credibility is gone.

001 directly addresses these issues.

Customers who have used the 001 technology have reported 4 or 5 to 1 in productivity the first time they developed a system with 001. These developments did not capitalize on repeatability and they were performed without the help of 001 experts. When 001 experts were involved on first time customer development efforts productivity approached 10 to 1. Our own more recent experiences have surpassed these numbers on first time projects, approaching 20 to 1. This is due to the fact that more 001 experts were involved and that HTI continues to enhance 001 with features which increase productivity even more than before. We continue to learn from our own use of 001, as developers, and pass this experience on to our 001 customers.

When follow-on projects take place, especially within the same type of application (e.g., bank project follows bank project or missile project follows missile project) the increase in productivity on a particular development effort is significantly greater and continues to increase with each new round of development of a system of the same or similar type.

Your Development Life Cycle will Change with 001

There is a marked difference between developing models, simulations and software with 001 and its automated tool set, and developing them with conventional techniques. The comparison chart summarizes some of these differences in comparing actual experiences on APOLLO with experiences using 001.

Whereas before 73% of the errors were interface errors, with 001 none are. Whereas before most interface errors were found after implementation, now all can be found before implementation. Whereas some of these errors were found manually and some were found by dynamic runs, with 001 all are found by automatic and static analysis. Whereas before some of these errors were never found, with 001 these errors are always found.

Whereas before requirements were bound to be inconsistent, they are now always consistent. Whereas before documentation and programming were manual, now documentation and programming are automatic. Whereas before there was no guarantee of function integrity after implementation, now there is a guarantee that the implemented form of the system is consistent with its requirements.

Other properties are inherent within 001 systems as a result of the theory embedded within them. An example is the ability of 001 systems to be flexible enough to respond to changing events and to reconfigure based on these events. Such properties are important in a dynamic manufacturing environment (e.g., dynamic scheduling).

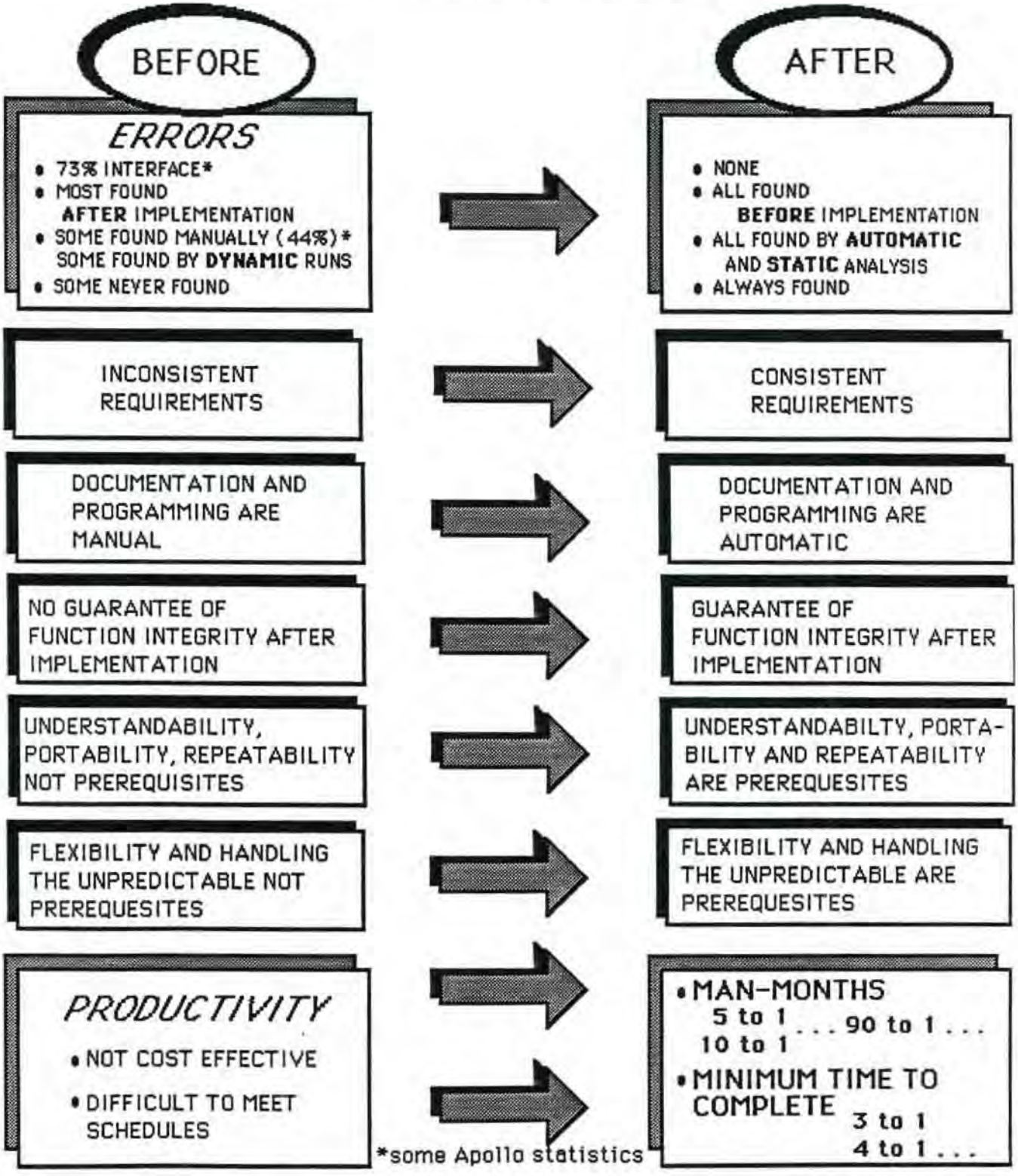
Another example is that 001-built systems can capitalize on repeatability to the greatest extent possible.

As a result of all of the above features, productivity increases significantly with the use of 001. Real numbers have shown productivity on first time projects (i.e., without capitalizing on repeatability) starting at 4:1 to 20:1 in man-months and starting at 3:1 to 7:1 in minimum time to complete. And of course, with repeatability, these numbers increase dramatically.

001 is your Edge

001 has been used to model factories; it has been used to model a system of human behavior; it has been used to develop software for missile systems for applications like SDI; it has been used to develop simulations for discovering oil as well as the simulator which drives them; and, it has been

A Comparison



used to develop itself. 001 has been used to develop asynchronous, event-driven real-time systems which can be processed concurrently and which communicate with each other.

Having a reliable system development process resulting in reliable systems is important in its own right, but the man-months saved and the shorter development times possible as a result of this reliability establishes 001 as a product which makes a dramatic impact in your organization's productivity.

Every hour you wait wastes time and dollars.

HTI Will Help you Prove it to yourself with 001

HTI has a special program whose focus is to help you develop reliable systems. HTI will develop a system of your choice for you using 001 or HTI will develop a system with you using 001 in order that you will become more thoroughly and more quickly acquainted with 001 and the system development techniques associated with it. HTI also has a training course showing you how to think and model using the technology behind 001 along with the 001 product. We want to share with you our knowledge and excitement in the use of this powerful product.

Your success is our success.

Our success is your success.

To learn more about 001, write or call:

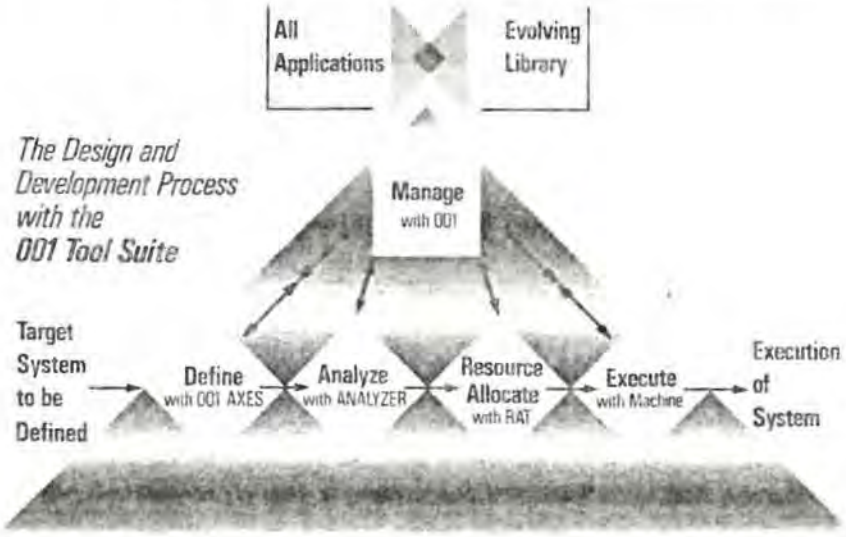
Hamilton Technologies, Inc.
17 Inman Street
Cambridge, Massachusetts 02139
(617) 492-0058

Product Name: 001™

Function description: Integrated tool suite for automatically developing ultra-reliable models, simulations and software systems.

001 is based on a philosophy that a reliable system is developed in terms of reliable systems. 001 supplies the building blocks to build a reliable system; as each new system is created by the user, it, in turn, can be used as a building block.

A 001 model captures all known properties of a system at a given time. It consists of an integration of functions and data types for a target system (for example, a real-time, asynchronous, communicating, distributed system) which resides on a combination of hardware, software and humanware environments. Once a model is defined, 001 will generate complete source code for that model. Rapid prototyping or production can then proceed; the model is simulated to observe its behavior in various dynamic states; software portions of the model become fully implemented system(s).



With its language, 001 AXES, 001 embodies a hierarchical structured network modeling approach based upon a formal concept of control. A model is decomposed using a control map which ensures that the relationships of all objects, in all states, are under control. AXES is both a functional and object oriented language. Functions are decomposed with an FMap control map. Data types are decomposed with a TMap control map. The FMap is used to connect functions with objects as input and output which abide by the rules of the data types in its associated TMap.

At any level of construction, a model may be submitted to 001's Analyzer. The Analyzer ensures that both data and functions are used in a consistent and logically complete manner, eliminating approximately 75% of all system errors.

Once a model is decomposed to the level of existing libraries and it has been successfully analyzed, it may be handed to the Resource Allocation Tool which will generate logic flow from both the function and the data control maps. This logic is automatically connected to previously existing function and data primitives in the core library as well as, if desired, libraries developed from earlier implementations. The generated source code can be compiled and executed on the same machine on which 001 resides or it can be ported to other machines for subsequent compilation and execution.

Because of its features of reliability, automation, abstraction and reusability, 001 maximizes productivity. Systems can be designed, developed and maintained with minimum man-months in minimum time.

001 contains the following components:

- An editor for supporting a user to define his system graphically or in textual form
- An executable specification language for defining logically accurate models that are consistent and logically complete for both functional and object oriented hierarchies and their integration
- An Analyzer for automatically detecting errors according to a formal set of rules
- An abstract type generator which generates a system of data types for a particular application domain from an object type hierarchy that is decomposed in terms of parameterized types
- A multi-language source code generator (RAT) which produces code of the user's chosen form (e.g., C)
- A documentor which produces a system definition, its implementation and its description

Margaret Hamilton:

A 'Higher Order' Has Been Her Calling

Apollo astronauts landed on the moon with her help. Now she is applying software expertise to her second startup.

Margaret Hamilton's latest startup — Hamilton Technologies Inc. — is 'headquartered' among the mixed splendors of her own Victorian-era home. Terminals and file cabinets are scattered amidst antique furniture and an eclectic range of art-objects. The splendor isn't just decorative — it is in a sense functional, Hamilton explains...

MHT: You didn't come out of a "typical" software background...

HAMILTON: I started college at the University of Michigan and was there for half a year. That was where my father went to college. Then I transferred to Earlham College in Richmond, Indiana. It is a Quaker school and that's where my mother went to college and most all my relatives, for that matter. I got my Bachelor of Arts there and that was it. I left from Earlham and came up to this area. I was already to go to Brandeis and continue with my advanced degree. Then I got involved with this stuff called programming. This was before computer science was even a field!

MHT: Was programming even a familiar term?

HAMILTON: Things like memory and programming were new terms. Most people were doing things by hand on simple calculating machines. There were a few people working with these things called computers. It was not a big thing at the time. I got involved with programming a very small machine with an MIT professor and I just never went back to doing anything else because I was really 'getting into it'. I went from working at MIT to doing a little bit with the SAGE system over at Philco-Ford, and then I heard about the Apollo project.

I just wanted to work on it!

I don't know why, I just wanted to! So I went over to the Instrumentation Laboratory of MIT (now the independent C.S. Draper Lab).

MHT: At what point did the lab determine that it would be involved with the program?

HAMILTON: It was quite early in the 1960s. The program started off with the unmanned missions and then progressed through to the manned missions. It wasn't long before I was running a group of people because I was one of the first people in on the project. The newcomers, sometimes, had been there only two weeks less than I had. Eventually I ended up heading the systems software area, which was providing operating systems — the glue — for the hardware that we put the applications into. It wasn't too long after that that I ended up running the whole software project for the onboard LEM (Lunar Excursion Module) and for the Command Module.

MHT: Were the basic designs of the the system already laid out when the project started or was technology advancing so rapidly that there were significant changes along the way?

HAMILTON: People were used to packing things in pretty tight at the time. If we were to look at the size of the Apollo Computer we had at the time people today would say it was impossible. Today we aren't used to sharing, layers deep, the storage that is used. We just found a way to shoehorn things in. Of course that made it much more complicated a task — more vulnerable to errors and that sort of thing. We did anticipate that we could do the job. But these projects are notorious for taking longer than estimated and more room and all that. We also had to iterate and reiterate and go back to the drawing board a number of times to adjust.

MHT: What programming language skills did you come into the project with and what role exactly did you play in the end product?

HAMILTON: I started off working in the operating system area and the system software area. Then the whole software effort — meaning everything from the operating side of the house.

I was involved with all of them. I then took over the command module software, and the LEM. So I ran all the onboard software effort.

I guess that I was young enough so that I wasn't aware of the magnitude of what I took on.

MHT: At that point in time did you feel you faced any gender-based discrimination?

HAMILTON: When I got involved with this whole thing it was before "female liberation". There were certain exceptions for some females who became almost like mascots or buddies, to the men. When I was in college I took math and physics and I was the only girl in the class!

I was used to it.

MHT: So the existing social fabric was able to stretch to accomodate you...

HAMILTON: Yes. Every once in a while there was an exception made and I was the exception. I guess when you get used to that you play the role more easily.

But it is true that when the person that I worked for brought up my name to run the whole division there were objections raised at first.

The word was that I was the most talented and the most appropriate one technically but they were afraid that some of the men might leave if I became the big boss.

"They were afraid that some of the men might leave if I became the big boss."

MHT: Did that end up happening?

HAMILTON: No. In fact I had the group for about 10 years and I think the only person that left went because *her* spouse was moving off somewhere. It was a very stable environment.

It was almost like a family environment. We had about 100 people at the time.

I don't think it was really to my credit as much as it was the Apollo project and its being so exciting. People were there night and day and there was a sense of mission.

So I had the advantage of that going for me. It was just a very unusual circumstance and very exciting. I still look back on it as a very exciting time.

Directly, just over 100 worked for me. But indirectly there were a couple of hundred people.

MHT: Your college background included not only liberal arts at a Quaker college, but also plans for further studies at Brandeis...How did that prepare you for later work in software?

HAMILTON: At Earlham I took not only mathematics but a lot of philosophy and religion courses. Not because I was really into it as a focus but because the professors were so good and they were really good courses. I wanted to have exciting courses and I wasn't sure exactly which way I wanted to go.

When you look at the software environment, there are two kinds of disciplines. One is the more scientific kind and the other is more abstract. I think I have tended to be on the more philosophical and abstract bent. So the philosophy and religion environment — my father was also a philosopher — I think all of this had a lot to do with the direction I took.

I was trying to understand concepts and make new concepts out of it. And I was making mathematical theory as a result of empirical studies and then doing the whole analysis of it and wanting to abstract and draw from it — as opposed to just wanting to make something work.

MHT: Do you think the value of that kind of preparation is recognized in places like MIT?

HAMILTON: I do think that sometimes people get too specialized. Sometimes a place like MIT, unless you take a broad list of courses, can get you locked into being too specialized.

Sometimes you can get someone out of very small school in the middlewest and they can be very good at being able to handle many different kinds of situations. I don't know that it is so much a philosophy versus math or science issue as it is, really, whether you have the ability to abstract and think.

There are individuals you can pick off the street — it could be a carpenter with no college education — that could probably do this if they have the ability to create or abstract — or perhaps someone who is an artist.

It is a kind of freedom of thought. People at MIT are mostly specialists — though there are some there who are both specialists and good inventors and able to think this way.

MHT: How did the Apollo work conclude?

HAMILTON: In the Apollo project our mission, all our missions from the software point of view, were flawless. There were no glitches in the software area. I know for a fact, however, that if we had flown another 10 missions one would have shown up.

It was so complicated and we didn't really have an opportunity to test out the software before the fact. But, as a result of the experience in the Apollo flight software, I took people who worked for me, members of my staff, and I decided to do an analysis of our experience so that we could learn from them how to do things for the space shuttle.

In the analysis we keyed in on the management techniques and the development techniques looking at the software and the various errors and we looked in particular at the software errors that took place during the integration of all the modules. We looked at errors that occurred in the putting together of the guidance and navigation and control phase, as well as the input and output activities. When software people on a large project begin to take their modules and put them together

"In the Apollo project... all our missions were flawless...I know for a fact, however, that if we had flown another 10 missions one would have shown up."

they are going to have interface problems — they are not going to work together. That's where you get your most subtle errors. In a large project the majority of time is spent on testing that aspect of it. In the process of actually analyzing the actual errors of the software during its development we found that 73 percent of the errors were what we called interface errors. They had to do with ambiguity of logic — as opposed to user intent.

MHT: What are some examples of that?

HAMILTON: For example, you might want to have something work with certain radar data when in fact that entity would have no right to be working with that data.

Or you might have the timing mixed up so that one module wants to turn on an engine while another wants to give it data.

If I were going to go on Eastern Airlines somewhere and they gave me a seat and I found someone else there when I boarded — that's an example of an interface problem because the system specified both of us to the same location.

It could also be a timing problem because perhaps we should have had the same seat, but on different flights.

We found out that 73 percent of the onboard errors fell into this interface category.

MHT: How did you succeed in locating these errors?

HAMILTON: The project, adjusted to todays standards, was a $250 million project. Half of that was simulation to find errors...So we ran $125 million worth of simulation to find these bugs. In fact, it turned out, about half of the errors were found by manual processes.

Often people were not purposely looking for errors but ended up finding them when they were looking for something else.

We had one guy from TRW who was an independent verifier. He would go through the listings and find all sorts of errors — he was just amazing.

MHT: Did people spot errors because they knew the project so well that the error[illegible] stuck out?

H[illegible]ON: At the time we were [illegible] three year old daughter

(continued on page 16)

Hamilton

(continued from page 3)

was helping run a simulation of the astronauts' mission.

She was pushing the buttons in a way that the astronauts would not normally push them and she found a major bug . . .

The third statistic that we found is that 53 percent of these errors existed when the mission was ready to fly.

It was a combination of things that would make it show up and come out. As a result of those findings we started trying to find ways to get rid of the errors.

If we defined a certain something it would get rid of a whole class of errors. As a result we fell into this whole mathematical theory called higher order software.

The point was that if we used this theory in defining a software system we would be rid of these interface errors. Once we realized that we could do this we then realized that we could define an automated tool that could actually analyze the system definitions and tell you when you had used the theory correctly.

If you used the theory correctly you could define the system to be free of those errors. And if you didn't you could go back and fix it. We felt we could also build a tool that would look at that and automatically produce source code that would be free of those kind of errors.

MHT: When did this occur?

HAMILTON: Saydean Zeldin and I were still at Draper when we realized this. We got a lot of attention regarding the potential at DOD and NASA, so we decided to go out and start our own company, build the automated tools, and sell them to the marketplace.

So that was the beginning of Higher Order Software Inc., some 10 years ago.

MHT: Had you viewed starting the company with any trepidation?

HAMILTON: We were young enough and inexperienced enough that we didn't know any better.

MHT: How many founders were there?

HAMILTON: There were a few of us, but Zeldiln, in particular, had worked on this and worked for me at Draper.

We started the company with no investment. We learned to live on nothing. We violated all the rules.

We went into the government services business initially as a way to bootstrap ourselves.

MHT: Did you have any assurances of getting that kind of business when you left Draper?

HAMILTON: We thought we did. We were all set to get a million dollar project for our first year when we left. But the general who was going to sign the thing left and a new one was assigned.

As a result I flew down in something of a panic to try to salvage something. We ended up getting a $20,000 contract to get us going. It wasn't a million dollars but it was enough to eat, stretched out for a while. It was very tough to start a business.

MHT: Did you have any problems leaving Draper?

HAMILTON: They weren't happy that we left but they weren't happy with a number of startups that left Draper at that time. Part of the reason these things happened was that Apollo was starting to wind down and we all needed an encore. It was so exciting during that time — some people who were there from the beginning and were there for the end had to go do something else exciting. The management wasn't happy to see those key people going but they got over it and they are over it now.

Robert Duffy, the president of Draper, and I are very good personal friends now.

Eventually we worked out the fact that we had left.

MHT: The year 1976, when you started HOS, was a real low point for defense spending?

HAMILTON: Yes. There were a lot of layoffs along 128. I had to lay off some people in my own division at Draper, which was heart breaking.

MHT: And how did HOS fare?

HAMILTON: We started off doing only government work. And, even when we were very young and very small we started off being the contract monitor for number of larger firms such as Rockwell and GTE. We were managing them. We gave them the subcontracts, and we also worked with manufacturing efforts to bring in new design techniques for defining things broader than software. It's logic really. We were involved in all these projects and we were making profits for several years. During this time we were using the profits to begin building the automated tool set — we had been using the theory manually before. Once we got to a prototype stage where we could demonstrate the power of this thing we had investors, particularly Venrock, looking us up. They said they wanted to invest and at the time (about four years ago) we said we didn't want an investment. We sort of kept in touch with each other for a few months and then we decided we weren't really getting the attention in the marketplace that we needed. We felt that we needed financial and management support to take the company from a commercial product stage to actually putting something in the marketplace. That's when we decided to go with the investors.

MHT: How large had you grown at that stage?

HAMILTON: Perhaps $2-3 million annually. We weren't huge but we were chugging along.

MHT: How much was the investment?

HAMILTON: It finally totalled $15 million over several rounds of funding.

I spent a great deal of my personal time raising money so it was a whole different thing for me, running all kinds of things and the company.

We raised the money in order to build the commercial product and put it on the VAX, and also to be able to build it for many different environments — systems with Fortran, C, and Cobol.

MHT: Is there anything else like this?

HAMILTON: No. There are people who have done part of what it does. There are people who have come up with products that can produce code from requirements, just as you can produce code from a higher order language. There are also people who can define what the problem is but there is no one out there who can define the requirement free from error and produce code with the same integrity.

It hasn't been done to our knowledge.

MHT: Is that patent protected or proprietary?

HAMILTON: Patenting software is complex. The theory is published in the public domain — though most people don't seem to be able to understand it.

There are several challenges along the way. First of all you must be able to understand the theory. Then there is knowing how to take the theory and put it into practice. That's not published anywhere. What took years of thought is hard to replace.

Also, once you have the greatest technology in the world, if no one understands how to use it, it's no good.

If everyone only knew how to ride a horse and there was a jet plane sitting there it would be useless. So there is a whole technical marketing challenge — getting that out into the marketplace. At HOS there were several hurdles. The actual hurdle of putting the product out there and making it do what it was supposed to do, was one. We did it.

The problem of actually getting it into the marketplace was a challenge beyond what we had estimated it to be.

IBM had the problem with the database and Cullnet too. But taking that kind of technology and first of all

(continued on 17)

Hamilton

(continued from page 16)

having the marketplace understand the benefits of going through this and letting them know about it was the major challenge. Even if somebody knew the technology they still would have to go through the learning process of what you do with it.

So it takes a while for somebody to catch on.

MHT: You are going to be immortalized for your Apollo work by a famous museum, I understand...

HAMILTON: Two museums actually. The Smithsonian is collecting all material that had to do with the onboard flight software effort...I have a whole room of archives.

Trying to organize that is a full time job in itself.

The Fort Worth Museum is also doing an exhibit on me and my career.

MHT: Are there upper limits of what you can accomplish in terms of the scale of the projects?

HAMILTON: The product does a lot of things right now. Now at Hamilton Technologies Inc., we have a new generation of the product and there will be another one...you can always do more. You can always make it run faster or be more user friendly.

But largeness has nothing to do with the problem if you've found a way to handle smallness. If I find a way to define modules which are reliable and I find a way to hook together those modules in a way which is reliable then I have found a way to build big systems. In other words I can take primitives, if I'm assured they are reliable, and hook them together with primitives.

MHT: This technology is expected to have SDI applications?

HAMILTON: There are several issues with SDI. One is whether or not you can put modules together and have them be reliable. Another is the unpredictable things that could happen during a mission and what the software can do about it.

MHT: In other words a fault tolerance concern?

HAMILTON: Yes, partly fault tolerance both in the software and the hardware and the combination, because SDI talks about the software problems. But sometimes they are talking about the software problem because they don't know the definition of a system problem. It's a user intent area. In the world of systems people don't blame the hardware because its sitting there and its already finished, usually. Instead, they blame the software, if indeed it's going to be put into software.

One thing you have to do is decide whether the difficulty is one of understanding the problems or whether it is the software.

You get into ambiguities. You can solve that ambiguity problem with this technique. It does not solve the user intent problem. I am not going to guarantee that this will solve what you are doing. I'll only guarantee that what you put in there is consistent and logically complete.

So that's one area. Another is the fault tolerant part.

On Apollo we had unexpected problems. We had lightning hit the craft twice. The software was prepared to handle that. It was prepared to come back up after a disaster like lightning. Because of a checklist problem, the astronauts once hit the wrong switch which overloaded the computer. It didn't give enough time to the software and made errors happen in the landing on the moon. The software was prepared to handle that, too, because it was what we call an asynchronous software environment. It could handle unforseen events.

It's not that the software knows lightning happened — it knows the effect of lightning happening. Everybody thinks that (for SDI) you have to be able to predict all the things that can happen. That is too large a set to even talk about. But when you think about what can happen to the software itself — it is a much smaller class of problem.

So the other thing that has to be done is to have this kind of asynchronous software environment which can handle the unpredictable and be able to reconfigure. That would mean that something can then come in and the software will say — this is higher priority than anything I've got going and I'll put this other stuff on the back burner and let this emergency come in...

MHT: So each element of the system, then, would have some survivability?

HAMILTON: Right. The mistake that people in Apollo made and on the Mars mission and parts of SDI was, at least back in the old days, to say that the astronauts won't make a mistake. In fact the astronauts do make mistakes because they are human. What you've got to do is build your software to handle these kinds of cases.

With the proper system software set up in SDI and the proper ways of creating the modules without interface problems the issue becomes, what do you want to do. Now the problem is the user — he's got to know what configuration he wants — what kinds of radars and so on. It's a system definition problems.

So largeness is not a problem in that respect. You've got to have enough radar and its got to be able to run fast enough but that is not a system definition problem.

MHT: What is Hamilton Tech doing at present and where is the firm going?

HAMILTON: I guess you would say that Hamilton Technologies is about the next generation of product. There was the evolution from the Apollo flights, then the empirical analysis needed to advance the product, then Higher Order Software entering the marketplace.

Now, we are the company which is going to make the technology work on very large and sophisticated systems and make it really take hold as a way of doing business.

HOS has concentrated on selling the product as an off the shelf item. We are planning on going in as an expert team and either do a job for the customer or show them how to do the job using these techniques.

MHT: Is the new firm self-funded or venture backed?

HAMILTON: So far it is not venture backed and that is intentional. We would like to be able to do it by ourselves, if things go the way we would like them to. We might need investors if things move faster.

It is possible. We will certainly give it a try first. We presently have five full timers and a lot of part timers. There is an awful lot of 48-hour-a-day work going on here!

MHT: What do you do when you aren't working 48-hours-a-day here?

HAMILTON: Actually, I'm part owner of an antique clothing store on Charles Street in Boston. It specializes in Edwardian and Victorian clothing and clothing of the 1920s...It's a hobby.

Sometimes when things just get too crazy here, on a Sunday, I'll just go there and take over and send somebody home — and have such a great time!

MHT: Sort of like the slogan, "When the going gets tough, the tough go shopping?".

HAMILTON: (laughs) Someone was kidding me the other day because I was on the phoen with the Pentagon discussing C^3, and I got an interrupt from my receptionist. I started saying, "it's the one with the red beads," and then I went back and started talking about C^3.

That was an example of an asynchronous executive. The interrupt came in and I processed it!

Anyway, it has been a hobby for 25 years but I just went into the business this year. I even have some favorites that I could see actually putting into a museum someday! □

Towards Software You Can Trust

M. HAMILTON
Hamilton Technologies, Inc.

With today's conventional software development techniques, as size and complexity increase so does the probability that the system, when introduced into operation, cannot be trusted. This is despite an inordinate amount of testing and evaluation. And when a software system works, the cost of attaining such a state is often needlessly high. The predictable result is wasted dollars, lost time and missed deadlines. For many systems, coping with events that cannot be entirely predicted is vital to effective real-time system performance. For some systems, like those designed for national defense, the uncertainty of actual environmental conditions at the moment of truth can present challenges to operational reliability that border on the impossible.

A software technology which reverses these trends is needed. This requires a major changeover to effective automation. The human effort that has traditionally been required to transform software every step of the way, from concept formulation to operational status, needs to be redirected. It has been shown that the human element both limits productivity and fails to practice sufficient rigor in translating the specifications that have been established to control its activities. People should be concentrated on that part of the process that will always require intellectual effort: the derivation of specific requirements from high level statements of user need. If requirements are stated in formal and practical way, then it is possible to automatically translate requirements statements into lower level, machine readable, executable forms.

These forms should be of a selectable level of definition, appropriate to the current level of understanding of the system as a whole. Such a concept offers the possibility of a continuously testable realization of a candidate design, from its broadest definition to any desired level of detail, and ultimately to the level of object code for the processing architecture selected to implement the system. The intermediate levels of definition would serve as prototype developments to assist the definition of the final detailed designs. Since each realization is automatically derived from the preceding, the traditional errors of human translation associated with the phases of the conventional software life cycle are eliminated.

To provide these properties, the *same* technology is needed to correctly define and integrate *all* the relationships within a system and with respect to that system, including the system with which it is developed: function to data, function to function, data to data. The philosophy behind this technology is that a system can be reliable if it is constructed from reliable systems with reliable mechanisms. Once a system is constructed from reliable primitives it (the system) can be used to build other systems along with the more primitive mechanisms. Such a concept allows repeatability to be used to the greatest extent possible.

An automation of this technology would allow one to check for using it correctly. In doing so, an automated tool set can be used to design and automatically analyze system specifications for consistency and logical completeness, eliminating interface errors. It would then automatically produce source code from these specifications with the same level of accuracy as the validated specifications. A system would be designed with its specification language, analyzed with its analyzer and implemented with its resource allocation tool (RAT). Documentation would be automatically produced which corresponds to the specification and would be provided to the user along with its corresponding source code.

Inherent in this automated tool set is intelligence that is in the form of a primitive rule set that has been derived from system engineering "experts." Further, it would be an expert system for developing expert systems. Given an application expert's input, new rule sets can be automatically derived from the more primitive ones. This computer-aided thinking tool set would place a major focus on the design and on the automation of the test and evaluation of a system during that part of the software development process that experience shows is responsible for most software problems: functional specifications.

Such an automated technology is 001™, developed by Hamilton Technologies, Inc. (HTI). Its purpose is to provide mechanisms for generating trusted software systems. Trusted software systems are systems which are understandable; they capitalize on repeatability, they can handle change and the unpredictable; they can be developed within affordable budget and time constraints; they are systems with zero defects. To date the 001™ technology has concentrated on developing large, complex and flexible real time software systems which are free of interface errors. Interface errors commonly account for over three-fourths of total error found in a typical large software development process. The remaining errors fall into a class of errors associated with user intent. The 001™ environment continues to reduce the user intent class to a much smaller number as it "learns" from its user experts.

The 001™ technology has evolved from empirical studies performed by Margaret Hamilton for over 20 years, first as director of the on-board flight software for APOLLO at Draper Labs, then as founder and president of Higher Order Software, Inc. from 1976 to 1986, and now as founder and president of HTI. The size and complexity of APOLLO software system afforded Hamilton and her staff an opportunity to analyze virtually every type of software error. The result of this intensive analysis was a mathematical theory which has proven to eliminate the largest source of software errors: ambiguities of logic. This theory has been reduced to practice in the form of the 001™ technology and has been proven to work in large complex systems.

(Continued on next page)

HTI recently used its specification-based approach to produce a space tracking application in less than a third of the time that software metrics predicted for conventional approaches, with a more than 10-fold increase in programmer productivity.

The meeting of the Maine Computer Society is scheduled for 7:00 pm on Wednesday, April 22, at Merry Manor Inn at Maine Turnpike, Exit 5, on Route 1 in South Portland. A social hour will precede the meeting at 5:30 and dinner at 6:00 pm. For dinner reservations call John Andrews at 207-775-8368 or the IEEE message machine at 207-283-1310.

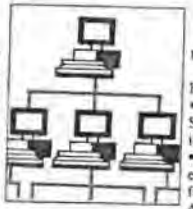
The Reflector

APRIL 1, 1987 VOL XXXV, NO. 6

PUBLISHED BY THE BOSTON SECTION OF THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS FOR THE IEEE CENTRAL NEW ENGLAND COUNCIL

SOFTWARE

Artificial-intelligence applications appear at last
Large-scale software projects gain tools
Strategic Defense Initiative presses on



Artificial intelligence, long viewed by many engineers as a back-burner field with mainly future possibilities, moved last year to establish itself as a practical tool. Applications proliferated, with emphasis on business and financial uses. Some engineering troubleshooting aids also emerged. A significant development was the spread of AI software capable of running on many different hardware platforms.

"Artificial intelligence is simply the next logical step in data processing," said Bill Johnson, vice president for distributed systems at Digital Equipment Corp., speaking during a press conference in Philadelphia in August at the annual meeting of the American Association for Artificial Intelligence.

AI was a software highlight in a year that also saw results flow from many of the software consortiums and government initiatives started in the last few years. In addition, the concept of reusable software made a strong showing in an unexpected segment of the market—personal computers.

But it was AI that yielded some of the more visible progress. As business applications proliferated, software companies took steps to make their product available on conventional computer systems as well as specialized workstations. PC-based AI tools now abound, and several high-powered aids for knowledge engineering are being recoded from Lisp to the C language to make them more acceptable in business environments. One expert-system tool has been written in the most business-oriented language available—Cobol.

Portability—the capability of running the same AI software on many different machines—was enhanced by greater use of Common Lisp. Although some computer companies have developed their own versions of the language, many are turning to third-party developers. Both Sun Microsystems of Mountain View, Calif., and Apollo Computers Inc. of Chelmsford, Mass., for example, use a Common Lisp developed by Lucid Inc. of Menlo Park, Calif. (Lucid's Common Lisp also is available for Digital Equipment Corp's VAX computers and others).

Xerox Corp., among others, showed applications ranging from factory simulation and capacity planning to an expert system for locating enemy radar installations on a battlefield. Under a marketing agreement reached last year, Artificial Intelligence Ltd. of Watford, England, will enhance and sell Xerox's Trillium software for designing control panels. Syntelligence Inc. of Menlo Park announced a multimillion-dollar pair of packages: one for evaluating bank loans and setting appropriate interest rates and collateral requirements, the other for evaluating insurance risks.

Intellicorp of Mountain View and Teknowledge of Palo Alto both showed enhanced versions of their knowledge-based system development tools. Some industry observers expressed concern that the enhanced tools now reaching the market run too slowly to be of much use, however.

To help remedy those speed problems new AI hardware was also announced:

- Digital Equipment Corp. presented its AI VAXStation, incorporating a MicroVAX II with a graphics display and a Lisp programming environment.
- Texas Instruments Inc. of Austin demonstrated early samples of its Lisp chip, built under contract from the U.S. Defense Advanced Research Projects Agency.
- Xerox announced a new Lisp processor, although it did not specify when the chip would go to market.
- Sun demonstrated the Sun 3-260, a 68020-based workstation with benchmark performance surpassing some Lisp workstations.

Programming gets larger

While companies that produce software tools for AI are concentrating mainly on increasing the power available to a single programmer, many projects are too large for one programmer, or even a small programming team. The term "programming in the large" has been adopted for these projects, which range from telephone switching systems to aerospace and military programs of many varieties. They generally have complex specifications and are designed to operate for 10 years or more.

Historically, such large-scale software projects have been prey to highly visible failures on first operational use—from a ballistic missile warning system of the U.S. military that registered moonrise as a Soviet attack, to bug-ridden early versions of IBM's OS/360 operating systems and finicky software that shut down the first launching of the space shuttle.

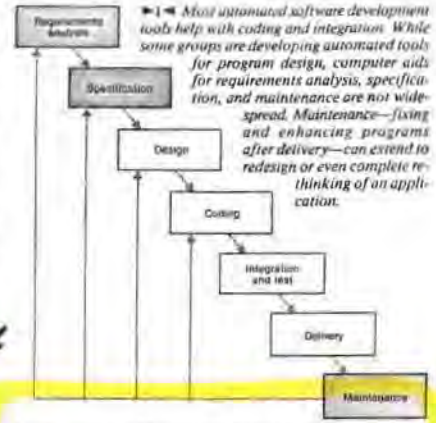
Furthermore, although software engineers long ago developed compilers, debuggers, and utilities for source code control to help automate the later phases of programming, automated tools for program specification and design have not been as successful. Instead, programmers have had to use manual methods—essentially diagrams of program function drawn on paper—that do not lend themselves to checking whether the program developed matches the program that was specified, or whether the specifications were correct in the first place [see "Zero-defect software: the elusive goal," *Spectrum*, March 1986, p. 43].

Tools for programming in the large include design and specification languages, coding tools, and integration tools. When dozens of programmers are working on hundreds of modules, it is imperative that the correct version of each module be used when the software is linked for testing or production.

Several organizations are working on software that can produce working code directly from specifications, an approach that improves software productivity and quality. First, the specification for a program is generally more concise and easier to understand and verify than the program itself. Second, since automated tools produce a testable program quickly, developers can determine rapidly whether the specification describes the job they want done.

Margaret Hamilton, president of Hamilton Technologies Inc. of Cambridge, Mass., and former head of flight software for the Apollo project, said her company had used such a specification-based approach to produce 17 000 lines of C code (not a very large program) for a space tracking application in less than a third the time that software metrics predicted for conventional approaches, with a more than 10-fold increase in programmer productivity. She commented that such techniques do not fall prey to the exponential growth of development time with increasing program size that plagues conventional development techniques.

Estimates like this are problematic, of course, since the time it takes to develop a given piece of software varies dramatically from project to project. Software measurement techniques are that small and medium-sized software-development organizations also use such tools.



►1◄ *Most automated software development tools help with coding and integration. While some groups are developing automated tools for program design, computer aids for requirements analysis, specification, and maintenance are not widespread. Maintenance—fixing and enhancing programs after delivery—can extend to redesign or even complete rethinking of an application.*

LORRAINE M. DUVALL: EXPERT OPINION

"More software engineers are beginning to recognize that the software problem is in fact a system problem."

Both software developers and their customers have become much more sophisticated in the last few years. As a result, the software market is becoming increasingly competitive. To stay in business, many software companies have been pushing to apply the software technologies developed over the last 20 years—like advanced programming tools, software measurement techniques, and artificial-intelligence tools. In the artificial-intelligence field, for example, the results of R&D work were made commercially available in the form of expert-system development tools for PCs and other microcomputers.

In addition, large software developers in defense and aerospace, like TRW, Westinghouse, Lockheed, and Boeing, are using high-quality software development environments with powerful tools for editing source code, managing multiple versions of code, producing object code, and debugging programs. Measurement techniques to quantify many aspects of software development are also very important. By keeping track of everything from lines of code written, changes made, and errors found, to the time spent executing individual subroutines, these techniques can improve software reliability and reduce development costs. The challenge of the next few years is to make sure

Many consortiums start to show progress

Initiatives bear fruit

During 1986, many of the major software initiatives and consortiums founded in the last few years started to show progress, and some of them even developed prototypes of useful products. In Japan and Europe, corporate, national and international efforts are focusing on developing programming environments.

The European Community's Esprit project has already begun work on its entry, the Portable Common Tool Environment. It is based on a set of definitions for the interfaces between software tools, so that any programs abiding by those specifications can communicate with each other effectively, regardless of who wrote them or what brand of computer hardware they are running on.

In the United States, the Defense Department's Software Engineering Institute developed a curriculum requirement for a master's degree in software engineering. This curriculum can be used not only as a basis for instruction in colleges and universities but also to define the content and boundaries of the field of software engineering.

The Software Productivity Consortium hired both a chief executive officer and a chief technical officer in 1986, and began building a staff of computer scientists and software engineers. At the Microelectronics and Computer Technology Corp. (MCC), software engineering researchers demonstrated prototype systems for improving software productivity to their shareholder companies. These demonstrations focused mainly on conceptual design, one of the earliest stages of the software life cycle.

Software initiatives also yielded some major disappointments last year, with cutbacks, delays, and redirection in both the DOD's Stars software program and the software office of the Strategic Defense Initiative. The armed services, which were to take a major part in the Stars program, failed to reach agreement with Pentagon planners on its direction and form. The SDI software-technology program was redirected and many projects were canceled because of differences between the SDI office and the Department of Defense laboratories involved in the projects.

A continuing problem for the software engineering community is the widening gap between the capabilities of new hardware and the ability to produce software that utilizes them effectively. For example, we have yet to develop software techniques for parallel and distributed architectures. The Eastport Study Group, which analyzed requirements for SDI battle-management software, recognized this gap as a major problem and recommended that the system architecture and the battle management software be developed in parallel so that the hardware that may eventually be chosen is best suited to software development.

More software engineers are beginning to recognize that the software problem is in fact a system problem. Some organizations, such as MCC and the Software Productivity Consortium, are working to develop tools that address system development as a whole rather than software only. In the future, software engineers and computer architects will have to work together to develop computing architectures that take into account the difficulty of producing large amounts of reliable software.

Lorraine M. Duvall is president of Duvall Computer Technologies Inc., Rome, N.Y., a recently formed company specializing in the development and transition of software technology for government and industry. She was previously director of research for IIT Research Institute in Rome, N.Y. Duvall is chair of the IEEE Computer Society Technical Committee on Software Engineering, a member of the Computer Society governing board, and chair of the steering committee of the International Conference on Software Engineering.

Paul Wallich Associate Editor

PC software tools lead the pack

Software engineers in both academia and industry have for years advocated development of reusable software modules to improve programmer productivity and reduce errors. Many of their proposals, involving massive databases primed with module descriptions and artificial-intelligence routines to extract those modules and combine them into working programs, have yet to see the light of day. In the personal computer industry, however, libraries of reusable software abound, albeit in simpler form.

To develop an integrated text editor, spreadsheet program, telecommunications package, and database-management utility with graphics and multiple windows, the PC programmer can choose from at least these resources:

- Three communications libraries.
- Five graphics libraries.
- Two text-editing modules.
- Nine window libraries.
- Two mathematics libraries.

These libraries consist either of source code in C, Pascal, or some other language, or of object-code modules that can be linked into a program after the rest of the program has been compiled. In some cases, the program itself may be only a skeleton consisting of some screen displays, a few conditional branching statements to control program flow, and a host of calls to the appropriate library routines.

One of the most widely hailed packages of the last year is Dan Bricklin's Demo Program, written by one of the authors of the original Visicalc spreadsheet, a program credited by many with touching off the microcomputer revolution in business [see "PC pioneers: Visicalc," *Spectrum*, May 1986, p. 68]. The Demo program, available from Bricklin's company, Software Garden, Newton Highlands, Mass., lets the programmer create a series of screens that can be displayed in any sequence. For each screen, the programmer defines what keystrokes will trigger the display of another screen. Different keystrokes can trigger the display of different screens, thus mimicking what a user would see when operating a functional version of the application being prototyped.

Since the Demo program can also capture screen displays from other programs, it is useful for creating training materials. More recently, however, the Oakland Group Inc. of Cambridge, Mass., developed a utility called demo2c, which takes display sequences developed with the Demo program and turns them into C source-code skeletons. A programmer need only add the program code—perhaps in the form of calls to purchased library routines—to perform the desired functions.

Reusable software for PC programmers serves mainly to improve the coding process. However, it has only a slight effect on earlier stages, like conceptual analysis and program design. While tools like the Demo program may substantially improve the detailed design of a program, by allowing fast prototyping and revision, the overall program design may be constrained by the coding tools available. —*PW*

now receiving greater scrutiny, said John Musa, a researcher at AT&T Bell Laboratories in Whippany, N.J. Musa cited the use of statistical models to establish software reliability. For example, if a program under development is failing at a certain rate, models can help estimate the future failure rate and how long it will take to reach a given level of reliability. Software managers can thus make sure that programs are reliable enough for production without spending excessive time fixing the last few bugs.

A shakeout for consortiums?

While 1985 was labeled the year of the consortium for the many software groups that were formed [see "Laszlo Belady: expert opinion," *Spectrum*, January 1986, p. 44], 1986 was the year that many consortiums got down to business. They hired staffs, set directions, and began work.

At the Microelectronics and Computer Technology Corp. (MCC) in Austin, Bobby R. Inman resigned as chief executive officer at the end of the year to join a high-tech holding company; a successor has not yet been found. In the meantime, MCC's software researchers delivered both algorithms and software prototypes to the organization's shareholders. The software included extensions to the Prolog language for object-oriented programming and a software tool for analyzing Petri nets—graphs that depict the concurrent flow of information through a software system.

According to MCC software technology program director Laszlo Belady, research focuses primarily on the earliest stages of software development, "where requirements are a collection of fuzzy ideas." By aiding the exploration of alternate software designs at the earliest stages ▶|◀, MCC's researchers hope to improve the quality of large-scale software systems significantly.

At the Software Engineering Institute, Larry Druffel replaced John Manley as director. Druffel, who had earlier been head of the Ada Joint Program Office in the U.S. Department of Defense, moved to the institute from the Rational Corp., a new company that sells Ada hardware and software. The institute's mission is to improve the transfer of software technology from research to industrial use.

The Software Productivity Consortium, a group formed in 1985 by a number of large defense and aerospace companies, is still building up staff and developing its technical plan. Some elements of the plan include prototyping and reusable software. The consortium plans to develop tools for requirements analysis and program design and to develop "information processing libraries"—databases from which program code for a particular application can be constructed automatically.

SDI endures fire

Software reliability measures, always a matter of concern, were a special area of contention last year for what could become the largest software project in the United States: the Strategic Defense Initiative. Critics continued to argue that writing software for the project, popularly known as "Star Wars," would be so complicated that performance during an attack could not be guaranteed. The SDI office also continued funding software research as well as soliciting bids on a national test-bed facility for simulating strategic defense systems.

The SDI office realigned its software research program, moving funds from Department of Defense laboratories, which had been performing significant amounts of research, to SDI contractors and the Defense Advanced Research Projects Agency, which also administers the Strategic Computing Initiative. Some of this realignment was recommended by the Eastport Study Group, computer scientists convened by the SDI office. The group's December 1985 report recommended that the SDI office focus on architectures for strategic defense systems that would be possible to program.

While early SDI architectures had focused on a single, integrated system to keep track of attacking missiles and decoys [see "SDI: the grand experiment," *Spectrum*, September 1985, p. 34], the committee concluded that writing such a single enormous program correctly would be impossible.

The group's scientists concluded that an effective SDI system could not be built with current software procurement and development techniques, which it characterized as "decades behind the state of the art." It recommended strongly that the design of any SDI system start with software feasibility and testability, rather than with hardware—letting software development capabilities constrain hardware designs rather than the other way around. However, the publication last year of detailed plans for system architecture, with software research still in its formative stages, indicates that this recommendation may not be followed. ◆

TO PROBE FURTHER

MINIS AND MAINFRAMES—The theory of data-level parallelism and the operation of the Connection Machine are described in *The Connection Machine*, by W. Daniel Hillis, MIT Press, Cambridge, Mass., 1985.

PERSONAL COMPUTERS—Both *IEEE Micro* and *IEEE Computer* magazines frequently include articles relevant to personal computers; both magazines are available from the IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, Calif. 90720 (telephone 714-821-8380) or the IEEE Service Center, 445 Hoes Lane, Piscataway, N.J. 08854-4150 (telephone 201-981-1393).

General information on personal computers and their applications is provided in "Personal computers: lessons learned," a special report by John Voelcker, Paul Wallich, and Glenn Zorpette, *Spectrum*, May 1986, pp. 44-75.

For more detailed information on copy protection and the rights of software users, see "How disks are 'padlocked' " by John Voelcker, *Spectrum*, June 1986, pp. 32-40.

Personal computer emulators such as the Commodore Sidecar are covered in "Making your PC behave like another," by John Voelcker, *Spectrum*, October 1986, pp. 61-66.

The criteria for selecting the appropriate bus standard are described in "A framework for computer design," by W. Kenneth Dawson and Robert W. Dobinson, *Spectrum*, October 1986, pp. 49-54.

Compcon Spring '87, sponsored by the IEEE Computer Society, will take place Feb. 23-26 in San Francisco. For more information, contact Compcon Spring '87, Glen Langdon, Conference Chairman, IBM Corp., Dept. K54-802, 650 Harry Rd., San Jose, Calif. 95120-6099; telephone 408-927-1818.

The IEEE Computer Society will sponsor Infocom '87, to take place March 30-April 2 in San Francisco. For more information, contact IEEE Infocom '87, P.O. Box 639, Silver Spring, Md. 20901; telephone 301-589-8142.

News and current events in the personal computer industry are covered by two periodicals: the weekly *InfoWorld*, published by CW Communications Inc. in Menlo Park, Calif., and the monthly *Byte* magazine, published by McGraw-Hill Inc., New York, N.Y. Technical information on MS-DOS systems and software is available in *PC Tech Journal*, published monthly by Ziff-Davis Publishing Co., New York, N.Y.

SOFTWARE—The major source for information in artificial intelligence is the American Association for Artificial Intelligence in Menlo Park, Calif. Conventional software engineering issues are covered in the publications *IEEE Software*, *IEEE Transactions on Software Engineering*, and *Communications of the Association for Computing Machinery* (ACM).

Efforts to develop error-free software are described in "Zero-defect software: the elusive goal," by Margaret H. Hamilton, *Spectrum*, March 1986, pp. 48-53.

Conferences that focus on software include Compsac, held each fall in Chicago and sponsored by the IEEE Computer Society, and the International Software Engineering Conference, sponsored by the Computer Society and the ACM. The next ISE Conference will be held March 29-April 2 in Monterey, Calif.

DESIGN AUTOMATION—*Spectrum* last surveyed design automation technology in January 1983, p. 55. *IEEE Circuits and Devices* magazine devoted its July 1986 issue to workstations, and in September 1986 *VLSI System Design* published an extensive survey of CAD systems for integrated-circuit layout.

COMMUNICATIONS—A progress report describing the ISDN network being installed in the U.K. by British Telecom can be found in "The United Kingdom digital network" by British Telecom, *Telecommunication Journal*, Vol. 53, no. 6, June 1986, pp. 314-16.

Photodetectors and their importance to fiber-optic communication systems are described in "Optical detectors: three contenders," by Steven R. Forrest, *Spectrum*, May 1986, pp. 76-84.

Japan's advantage over the United States in the telecommunications products trade is examined in "Assessing Japan's role in telecommunications," by the *Spectrum* Staff, *Spectrum*, June 1986, pp. 47-52.

The IEEE Communications Society will cosponsor the Conference on Optical Fiber Communication Jan. 19-22 in Reno, Nev. Contact the Optical Society of America, 1816 Jefferson Place, N.W., Washington, D.C. 20036; telephone 202-223-0902.

The International Switching Symposium, sponsored by the IEEE Communications Society, will be held March 15-21, 1987, in Phoenix, Ariz. For more information, contact Ed Glenner, GTE Network Systems, 2500 W. Utopia Rd., Phoenix, Ariz. 85027; telephone 602-582-7792.

Details on mobile communications allocations can be found in "FCC Allocates More Mobile Frequencies" by Benn Kobb in *Personal Communications Technology*, July-August 1986, p. 17.

SOLID STATE—Details of the semiconductor trade agreement between Japan and the United States can be found in "U.S. applauds chip pact with Japan," *The Institute*, October 1986, p. 1. For a copy of the agreement as published in the *Federal Register* (Vol. 51, nos. 149-61, August 4-20, 1986), contact the U.S. Government Printing Office, Washington, D.C. 20402.

The latest technical details on solid-state circuits are covered in the proceedings of the 1986 International Electron Devices Meeting, held December 7-10. For a copy contact Melissa Widerkehr, Courtesy Associates Inc., 655 15th St., N.W., Washington, D.C. 20005. The record of the Custom Integrated Circuits Conference, held May 12-15, 1986, is available from Laura Silzars, Convention Coordinating, 6900 S.W. Canyon Dr., Portland, Ore. 97225. Introductions will also be made at the 1987 International Solid-State Circuits Conference, to be held Feb. 25-27. To register, contact Lewis Winner, 301 Almeria Ave., Coral Gables, Fla. 33134.

INSTRUMENTATION—Instrumentation issues—including techniques for built-in self-test, trends and tradeoffs in test economics, designing for testability, and software solutions to testing challenges—were addressed at the International Test Conference, Sept. 8-11, 1986, in Washington, D.C., sponsored by the IEEE Computer Society and the IEEE Philadelphia Section. For information about the proceedings of the conference, write to

Spot photo sources

Three of the articles in this issue open with photographs of relevant technology. The photos and their sources:

- Minis and mainframes—simulation of activity within the Connection Machine (Thinking Machines Corp.).
- Aerospace and military—false-color image of Uranus (Jet Propulsion Laboratory).
- Medical electronics—three-dimensional image of a woman's skull, based on computerized tomographic scans (Cemax Inc.).

0018-9235/87/0100-0071$1.00 © 1987 IEEE

Zero-defect software: the elusive goal

It is theoretically possible but difficult to achieve; logic and interface errors are most common, but errors in user intent may also occur

In October 1960, shortly after a new radar network to warn the United States of missile attacks had become operational, a radar station in Greenland reported the appearance of a massive attack—a large number of radar returns coming over the eastern horizon. The real cause of the alarm: the moon was rising.

In late 1985, as activity in financial markets escalated, the operations of one financial services company were brought to a halt as its computers reported error after error. The designers of a bond-tracking program had built room for only 32 767 bond issues into their tables, and the 32 768th had just appeared.

In 25 years, although the speed, memory capacity, and reliability of computer hardware have increased manyfold, the reliability of computer software has not. Certainly software has become more reliable, but bugs still crop up in programs of all kinds, from the smallest game on a micro to the largest operating system on a mainframe.

As the Government proposes to build immense real-time systems like antimissile shields, which require enormous amounts of trouble-free software, critics question whether such systems can ever be made to function reliably. Although software development methods have improved measurably in the last several years, error-free software is, in the opinion of most software engineers, an impossible goal.

But some software developers believe complex software can be developed that approaches zero defects by using formal specification techniques and computer-based tools. These tools first check the consistency and logical completeness of a set of formal specifications and then generate program code that matches them. Checking the specifications for completeness and consistency eliminates errors of logic that arise from oversights, while tools that produce code directly from the specifications eliminate errors that might arise in implementing the specifications by hand. Software developed by such techniques may not always end up doing what the user wants, but it will do what the user asks it to do.

How software errors occur

Most software development techniques proceed from requirements to specifications to designs to program code, using people to carry out the transformations from one level to another. Only the final step in development—generating machine code—is usually done by machine. In the other steps, two kinds of errors arise: those in which the user's intent is recorded incorrectly—like a misplaced comma in one National Aeronautics and Space Administration (NASA) program that sent a Voyager spacecraft toward Mars instead of Venus—and errors in which the wrong intent, considered in some larger context, is set down in logically complete and consistent fashion. An example of the latter might be billing programs that send threatening letters to customers who owe $0.00.

Many techniques have been developed to deal with the first kind of error—incorrect statements in software—especially at the program-code level. Compilers can check the syntax of statements submitted to them, and "strongly typed" languages can enforce consistency between different uses of the same variable. A variable defined as an integer in one place, for example, cannot be used for character operations somewhere else. These remedies are static methods for software verification, which work by examining program source code rather than by testing a program's execution.

Static methods can also be used to check program specifications, or any other formal representation of a program, provided those specifications have been written in machine-readable form. Recently tools have begun to appear that can check program specifications for inconsistencies, ambiguities, and incompleteness in the same way that compilers check the syntax of program code. But static methods cannot eliminate all errors from either code or specifications; in particular, they cannot deal with errors of user intent.

Errors in user intent are the hardest to catch, because a program containing them can be consistent and complete but still give the wrong results. Some errors of intent arise from oversights—the equivalent of typographical errors in program code—while others come from a genuine confusion on the part of the user as to what the program should do.

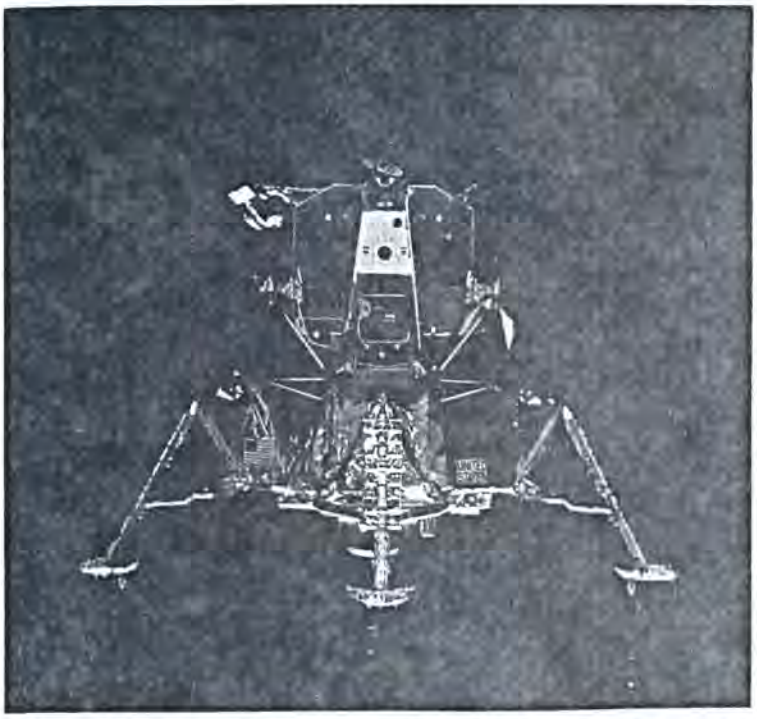
An additional complication is that software is almost always part of a larger system that also includes hardware—and humans. The software can be reliable and free of defects, but the

Defining terms

Interface error: an error that occurs because of improper use of a program module; for example, a module might be given too many arguments for input, or the arguments might be passed in the wrong order.
Primitive operation: a procedure that cannot be broken down into other operations; depending on the application, a primitive operation in a specification may translate to only a few machine instructions or to an entire software subsystem.
Specification: a formal description of what a program will do, phrased in terms of its inputs, its output, and the relationships between them, rather than in procedural form.
Strong typing: a characteristic of some programming languages that enforces constraints on the use of variables to reduce mistakes; for example, a strongly typed language would not allow a variable of the type *apple* to be added to one of the type *orange*, even if both types were represented as integers.
User-intent error: an error that occurs because the user did not properly think through a problem before committing it to software.

Margaret H. Hamilton Hamilton Technologies Inc.

  0018-9235/86/0300-0048$1.00©1986 IEEE



NASA

Software alarms caused by improper actions of the users nearly prevented the first moon landing. Software reliability is contingent on a system definition that includes the hardware and human parts of the loop as well as the instructions to the computer.

system that contains it can still fail, and in ways that resemble software errors. A few examples from the NASA Apollo program:

- Just before the Apollo 11's moon landing, the software sent out an alarm indicating that it was overloaded with tasks and had to reset itself continually to service critical functions. The program was still functional, so the landing went ahead. Later it was found that an astronaut had mistakenly been instructed to turn on a sensor that sent a continuous stream of interrupts to the processor, causing the overload.
- During the midcourse phase of another early Apollo mission, an astronaut keyed in the flight software program for liftoff, causing the computer to lose data. In later flights, checking software prevented wrong selections and made the overall system more reliable, but during one simulation, overzealous checks prevented the crew from entering any commands at all.

In any case, software should not be blamed for errors of user intent. An extreme example would be a user who wants to guide a rocket to the moon but inadvertently sets down a definition so that the software directs the process of baking a cake instead. If the software is still rational and unambiguous, is it the software's fault that it didn't guide a rocket to the moon or is it the user's fault for having somehow defined the problem incorrectly?

Fortunately, problems of incorrect user intent are only a small proportion of the total software errors in most projects. Far more common and more subtle are interface errors—those resulting from interactions between two software modules coded with different premises about each other's function and syntax, or between software modules and the rest of the system. Formal specification techniques and software tools for checking those specifications can eliminate inconsistencies in all phases of software development except the very first: they cannot guarantee consistency between the requirements that the user puts down on paper and those in the user's head [Fig. 1].

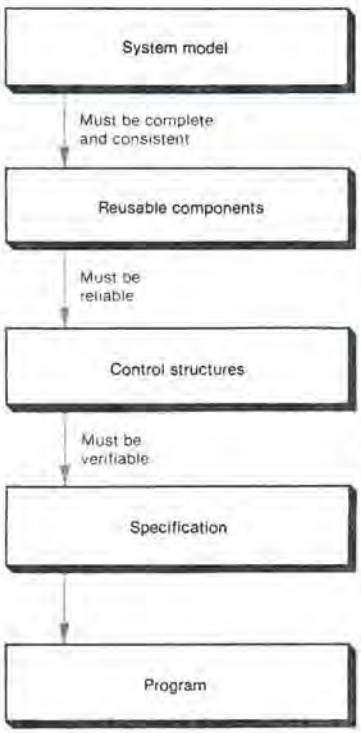
Improving software reliability

The many methods and tools used over the years to reduce errors in software systems run the gamut from trial-and-error testing of software code to methods for preventing certain kinds of errors. Methods for testing software after it is completed give the program a set of input values and then check whether that produces the correct output. Since it is impossible to check all combinations of inputs for most complex programs, software developers break up the program into modules to distinguish interactions between inputs, and they pay special attention to how a program handles both invalid inputs and inputs at the extremes of its range. But even after a program has been thoroughly tested, additional bugs often show up in use.

The kinds of software development practices used have a strong impact on reliability; good program design can reduce the likelihood of errors and can make the errors that do occur easier to find. Structured design, for example, reduces the interactions between program modules. This, in turn, reduces the chances for errors caused by clever programming tricks or by the propagation of bugs from one module to another.

A technique known as information hiding, developed by David Parnas, professor of computer science at the University of Victoria in British Columbia, is based on the idea that only the minimum information required about a module—what operations it performs and what inputs it requires—should be available to any other module. Changes can thus be made in the internals of a module without affecting other modules that make use of it. This approach has been codified in the separation of the specification and implementation of program modules in the Ada language of the U.S. Department of Defense. Reducing module size also improves reliability by making it easier to test individual modules.

Most methods for producing reliable software try either to prevent the occurrence of certain kinds of errors or to prevent those errors—once a programmer has made them—from finding their way into the final program code. Methods like syntax checking and type checking in compilers, unlike methods that eliminate certain kinds of errors, cannot prevent programmers from mak-



[1] Developing reliable software starts with logically consistent and complete specifications, which are decomposed into a set of primitive operations combined by means of control structures. Software tools at all levels ensure that the original specification is complete and consistent, and that the code generated matches those specifications. However, even software that is logically consistent and complete will not do what the user intended if the system model does not match the real world.

ing mistakes; they simply refuse to turn defective source code into machine language.

Logical proofs of correctness are another class of static method. Here the programmer makes assertions about the behavior of a program, and computer-based tools determine whether those assertions follow from the program code. Unfortunately, proof-of-correctness techniques have been used successfully only on small algorithms. The time required for a proof increases exponentially with the size of the program.

Software tools known as application generators or fourth-generation languages, used primarily to increase programming productivity, eliminate whole classes of errors by producing programs from a set of higher-level language statements. This is analogous to the approach known as "correctness by construction" in integrated-circuit design. For example, a database application generator might allow the programmer to specify a particular operation to be performed on all entries meeting some combination of criteria, while a standard programming language like Fortran or C would require the programmer to write out the iteration, fetch the variables in proper sequence from each record, make the appropriate tests, and branch to a similarly explicit routine to perform whatever operation was desired. A few lines of executable specification for an application generator or fourth-generation language can take the place of several pages of program code.

Some of the reliability benefits of application generator tools can also be attained simply by reusing program modules from one project in another or within a single project. Once a module has been debugged, using it again eliminates errors that might creep in if it is written from scratch. Of course the programmer must still supply the reused module with the proper data in the right format. Reuse of modules is in some ways a manual version of what application generators do by using chunks of code to implement the specifications given them.

In addition to improving reliability, the various methods of software reuse also increase productivity. They save time that would otherwise be spent in writing the same piece of code over and over for each new project or in debugging that code each time it was written.

Reuse also has its problems. Code that was written to fit many different problems may not be as efficient as code that is hand-tuned for a specific case. An application generator designed to produce one kind of program could prove unwieldy for producing a different kind.

General-purpose specification languages—sometimes called program-design languages—let programmers specify any kind of software system. These specifications can then be broken down either by hand or with automated tools to produce a detailed map of program modules and their relationships to one another. The map can then be turned into a working program. However, program design languages that rely on people to implement the final program may sacrifice reliability, because there may be no automated way to check the match between what the specification says and what the program code does.

A software tool called Refine, produced by Reasoning Systems of Palo Alto, Calif., can eliminate errors by going from specification to implementation. It uses a set of so-called rewrite rules to convert a program specification into an executable code. Rules govern the creation of loops for iteration and the specific implementation of abstract data types—such as the choice of a linked list, an array, or even a string of bits to represent a collection of objects.

Another tool is Use.It, developed to automate the methodology of Higher Order Software Inc. The techniques embodied in it are also used manually by some software developers. The Cambridge, Mass., company's tool gives the user a rigidly defined set of control structures for decomposing a top-level specification into modules until either primitive operations or preexisting modules are reached at the bottom of the hierarchy. The tree-structured specification is then checked for consistency and logical completeness and turned into program code in a conventional high-level language.

Software without errors

The two major aspects of methods for improving software reliability are reusing existing components and avoiding errors in the first place. The best way to build reliable software systems is to use components that have proved reliable, and to link them together with constructs that have been shown to be reliable as well.

The implication is that a system for creating reliable programs could be built by limiting the designer, and ultimately the developer, to design methods that are provably correct. The user of such a system would construct hierarchies of modules using only such methods together with reliable preexisting components, to develop a reliable system regardless of its size or complexity. Methods to develop zero-defect software should leave as few errors as possible for the most time-consuming part of the process: the dynamic testing phase of a system. All errors except those of user intent should be found earlier, through static methods.

The most basic target of static testing is ambiguities—inconsistent or logically incomplete sets of definitions—that can occur in software systems of all application types. Since there are methods that prevent ambiguities from being written, an automatic analyzer need only check for proper use of those methods. Other forms of static error analysis can be performed for specific classes of applications once the behavior common to members of that class is understood. For example, static methods could check that certain constraints—such as restricting dates to particular values—are satisfied in the use of a particular data type.

Once software tools have eliminated ambiguities, only performance errors—discrepancies between what the user meant and what he or she actually specified—remain for the testing phase. On nontrivial systems, over three-quarters of the errors that would occur with conventional software development techniques would be eliminated before testing if software tools for checking consistency were used.

Developing a program using reliable methods will almost certainly simplify the system definitions—an additional benefit that eases the mechanics of testing for any errors that may remain. Not only should there be far fewer errors to fix, but they should also be much easier to find.

Developing zero-defect software

There are three basic phases in developing a reliable system by the kind of methods outlined here: first the user develops a system definition, then software analysis tools check the definition for logical correctness, and finally a resource allocation tool produces source code in a language like Fortran, C, Ada, or Lisp from the definition.

The Use.It software tool has been successful in developing a number of software systems, including a control system for a large manufacturing plant that tied together shop-floor sensors, computer-based inventory management, and parts-handling equipment; a personnel management system for the U.S. Army; and a family of operating-system utilities for the applications-software division of a large computer company.

To develop a piece of software with Use.It, the top-level system definition is drawn with a graphics editor [see "Building a real-time system," p. 51], listing the inputs and outputs for each module in the definition, and the relationships between the modules are then defined. As with a number of other computer-based specification tools, a user can choose to work with either a graphical representation of the system or a text representation that is more concise but may not be as easy to understand.

In addition to defining the functional components that make up a program, the user also defines the data types the system will need. A series of axioms defines the behavior of each data type so that its use can be checked for consistency. The axioms define the primitive operations that can be performed on objects of a given data type, the relationships between primitive operations, and

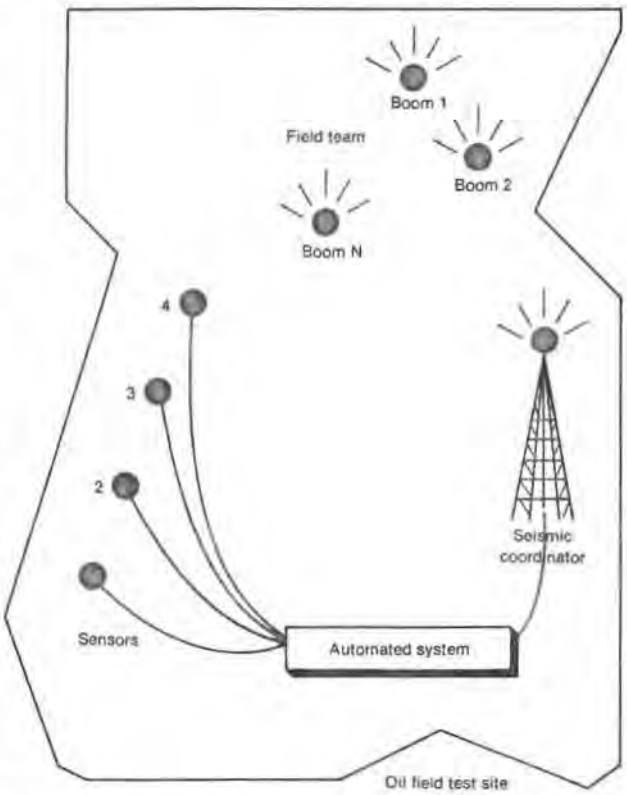
Building a real-time system from specifications

A program for controlling explosives and sensors for seismic oil exploration could be developed using software specification tools such as Use.It.

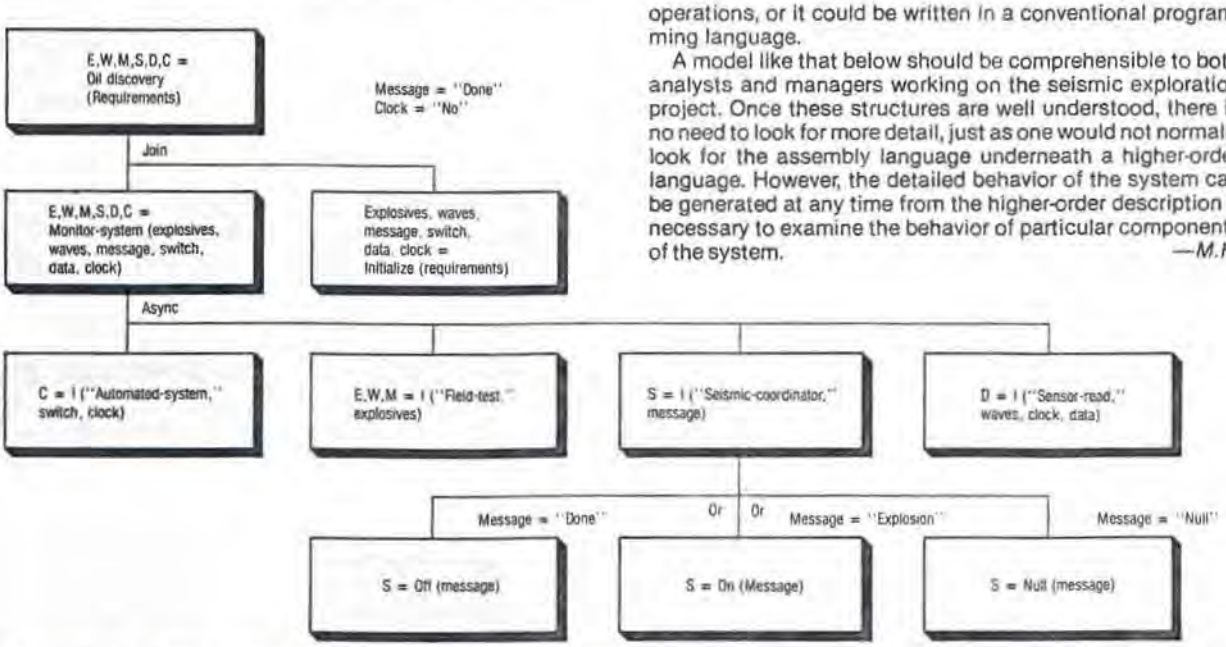
At an oil prospect, a field team drills a number of holes and loads explosives into them. Sensors are strategically placed to detect the seismic waves (right). The control program must activate the sensors to record data for 10 seconds after each explosion, but if another explosion occurs within that time the sensors continue recording. The automated system need not record the data; each sensor, connected to the automated system, has a microprocessor attached, to record the seismic waves. This process continues until the sequence of explosions is completed.

The system is put together using either primitive control structures or abstract structures derived from them [below]. For example, at the top level, *Join*, a primitive, transmits the user requirements to the initialization routine, which processes them and passes data to the monitoring module. The monitoring module passes data on the final system state back to the top level. The monitoring module is a series of functions controlled by an *Async* structure, which handles tasks that communicate with one another concurrently and asynchronously. Another primitive structure, *Or*, controls the subsystems of the Seismic Coordinator module. Depending on which message is sent to the module, it returns a different state of its output variable.

Since the process of decomposition for a system definition is not complete until a primitive operation for a defined data type is reached, the developer must check out the remaining tasks at hand at the bottom of the hierarchy. For example, since *Initialize* is not a primitive operation, either it must be decomposed further or an operation by that name must exist elsewhere for the system to be completely defined. If such an operation exists elsewhere, it could be built from primitive operations, or it could be written in a conventional programming language.



A model like that below should be comprehensible to both analysts and managers working on the seismic exploration project. Once these structures are well understood, there is no need to look for more detail, just as one would not normally look for the assembly language underneath a higher-order language. However, the detailed behavior of the system can be generated at any time from the higher-order description if necessary to examine the behavior of particular components of the system. —*M.H.*



The seismic profiling problem can be modeled as a tree of modules, with a function at each node and control structures forming the links between them. The graphic specification shown offers enough information for a computer to develop and execute such a system, assuming the existence of previously defined libraries of program modules to support this type of application. Here, requirements of the user are input to the seismic profiling system (top box); a final state consisting of a set of explosions, a set of seismic waves, a message, a switch, a set of recorded data, and a clock are its output. Each high-level module in the system definition can be decomposed into a set of lower-level definitions. For example, the Seismic Coordinator function is broken down into a choice among three alternatives, depending on the message received. This kind of system definition technique is applicable to any kind of system design—software or hardware—because the functions required in each module can be performed by either a person or a machine.

the results of those operations. It is particularly important to include operations that will produce error conditions, as well as those the software is expected to perform.

Software developers working with Use.It decompose systems into primitive functions using control structures based on three primitives: *Or*, *Join*, and *Include* [see Fig. 2]. Strict rules govern the way each control structure is used, guaranteeing the consistency and completeness of the specification.

Or is used to control decision-making. Its output is simply the output of one or the other of the two functions it controls, depending on the value of a decision-making variable. Both of the functions in an *Or* must take the same variables as input and deliver the same variables as output.

Join is used for controlling functions that depend on each other. The right-hand child in a *Join* takes its input from the parent and delivers its output to the left-hand child. The left-hand child, in turn, takes its input from the right-hand child and delivers its output to the parent.

Include is used to control modules that independently perform part of the function of the parent module. One part of the input is passed to the right-hand offspring, the rest to the left-hand offspring. Output from each offspring is passed back to the parent. More abstract control structures, including recursive ones, can be defined in terms of these primitive control structures as they are needed.

Although each control structure is unique, certain generic principles apply to all:

- A function at a given node controls only those functions at the level directly below it.
- Each function must produce an output.
- Each function must control where its offspring get their input. If offspring could take their input from anywhere, verification would become impossible.
- Each function must control where its offspring's output goes.
- Each function must either produce values of the correct data type or inform its parent if the values do not belong to the proper type.
- Although modules can process each input as soon as it arrives, they must maintain the specified order of overall execution with respect to functionality, priorities, and timing.

Violating any of these interface principles may appear benign in itself, or even necessary to improve system performance. But any violation makes it impossible to verify the system and could introduce subtle but fatal errors.

On the other hand, if the module interfaces do fit together—if functions do produce values of the proper data type, if the correct order of execution is maintained, if modules do not violate the order of the control hierarchy—then a program will be reliable. With this approach, two systems of vastly different size could be equally reliable, since each system would consist of reliable components that are integrated using reliable constructs.

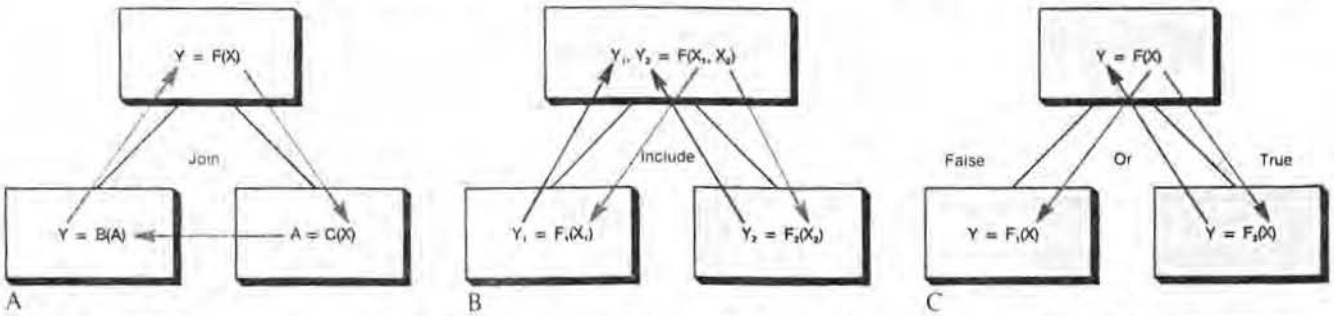
Formal definitions improve productivity

The main intent of these formal rules for constructing software systems is to eliminate errors before the fact, but the same rules that guarantee logical completeness and correctness bring other advantages as well. For example, because each variable is referred to in only one place, and only one function changes its value, all data flow is traceable, and the impact of each change in a model is known ahead of time.

In addition, each function in the hierarchy is assigned an unvarying priority (a parent, for example, always has a higher priority than its offspring). These properties allow the unambiguous allocation of computing resources to functions and processes. Because dependencies can be traced easily, multiple processors can be allocated as easily as single processors.

Software tools like Use.It also improve programmer productivity, because systems can evolve naturally from existing modules and less time is spent tracking down bugs. Productivity will increase the first time such a tool is used on a project, and it will continue to increase with each succeeding project in the same application area, because more application-specific subsystems will become available.

Productivity and reliability may increase even if such software tools are used only for parts of a software system, with other parts built by conventional methods. However, the full reliability gains of developing software automatically from specifications can be achieved only if these tools are used throughout the devel-



[2] Three primitive control structures can be used to construct reliable software systems from individual functional modules. Each module is defined by its output, the function name, and its input. In the example "Sum = Add (A,B)," the Join *primitive (A) controls functions that depend on each other; data flows from the parent down to one offspring, where operations are performed on the data. The new data is passed to the second offspring, which performs additional operations and passes the result back to the parent. The* Include *primitive (B) combines results from a set of independent functions. One offspring works on one part of the parent's data, and the other works on the rest. The results are passed back up to the parent. The* Or *primitive (C) makes a decision between two alternatives. Each offspring takes the same variables as input from its parent and produces the same variables as output. If the decision variable is true, then the right offspring passes an output back to the parent; if it is false, then the left offspring passes an output back. The data flow in the example shown is for the "true" alternative.*

All functional modules operate on objects of particular data types. In turn, axioms define the behavior of data types and the operations that can be performed on them. The ordered set, a list of objects of a single data type, is a useful type in its own right. For any ordered set containing elements of type T, *where* t *is an object of type* T, a *and* b *are Ordered Sets (of* T*), and* Nullo *is a constant Ordered Set (of* T*), the following rules hold: First (*Nullo*) equals Reject; Second (*Nullo*) equals Reject; First (Combine [*T,a*]) equals* t*; Second (Combine [*t,a*]) equals* a*; OEquals (*a,b*) means Equals (first [*a*], First [*b*]) and OEquals (Second [*a*], Second [*b*]).*

The following operations can be performed on an ordered set: First, Second, Combine, and OEquals. The first two axioms define the error conditions for an Ordered Set, the third and fourth axioms define constraints for the selection of elements out of an Ordered Set, and the last axiom provides a concept of equality for Ordered Sets. Ordered Set (of T*) is a parameterized type—*T *can be replaced with the name of any type.*

opment process. A single error in one primitive operation or the use of an external operation that has not been properly validated could compromise the entire software system. The wider the set of validated data types, primitive operations, and abstract control structures available, the less temptation there will be for software developers to compromise.

User intent still a problem

Methods based on formal definitions will help to approach zero-defect software, but they will not guarantee that the user knows what he wants to do. What they will do is help the user arrive at conclusions much more quickly than with conventional methods. Intent issues, not interface or logic problems, will be the only concern in testing software systems in the future. Furthermore, these methods would force the user to define systems much more clearly than is typical with informal specifications that cannot be checked with software tools.

Leaving the responsibility for user intent outside the domain of software in no way abandons responsibility for the most subtle errors. The most subtle and complex errors in the Apollo program and in other large and sophisticated systems have usually been interface errors, not errors of intent.

Even after interface errors and other ambiguities have been taken care of, performance testing to prove correctness of intent is not a trivial task. But it can be conducted on a more controlled and clearly defined module-by-module basis, in which boundaries are well understood and unwanted side effects are a phenomenon of the past. Indeed, most classes of errors that were previously discovered during performance testing no longer exist. Well-known interface problems, like the "deadly embrace"—in which multiple processes stall because each controls a resource that another process needs to do its tasks—are in fact eliminated by methods that synthesize code from logically correct specifications.

The question put forth by software critics should not be what methods can be found to produce defect-free software but rather when existing methods will become widely used. The good news is that the feasibility of complex software-hardware systems like those of the Strategic Defense Initiative (SDI) need not be dependent upon the inherent reliability or unreliability of software. The bad news, on the other hand, is that software can no longer cloud the issue or become a convenient scapegoat when a problem is not well understood. The real problem with large systems like SDI (and many smaller systems) is that the user needs to understand the specific application problem before it reaches the stage of a "software problem."

Application problems may be easier to solve if the requirements or specifications of a system—regardless of whether it will eventually be implemented as computer programs, hardware, or human systems—are defined in such a form that a computer and its software could find the ambiguities in those specifications. With such formal definitions, techniques for defining reliable software can be used to define reliable requirements for systems in general, since the problem in both cases is to define a set of logical statements unambiguously. Even when requirements can be computerized and analyzed for ambiguity, there is no way to guarantee that the user has put forth his real intent. But software tools and formal specifications offer a way to define exactly what the user said his intent was and to determine if it is consistent and logically complete.

If software tools are to be used to determine whether a set of requirements is logically consistent and complete, the requirements must be defined formally before implementation of the system begins. Far too often, this front end is treated casually. It is time to treat software seriously, as a science or engineering discipline from start to finish.

To probe further

Reliable software has been a concern of the computer industry for many years. The major themes of some conferences, including the IEEE's International Software Engineering conference, are productivity and reliability. Good periodicals on the subject of software reliability include the *Journal of Systems and Software*, published by Elsevier-North-Holland; *Communications of the ACM*, published by the Association for Computing Machinery in New York City; and *IEEE Transactions on Software Engineering*. The December 1985 and January 1986 issues of the *Transactions* were devoted specifically to software reliability.

System design from provably correct constructs, by James Martin (Prentice-Hall, 1985), discusses the process of developing correct specifications and generating code from them. The article "The relationship between design and verification," by Margaret Hamilton and Saydean Zeldin (*Journal of Systems and Software*, No. 1, pp. 29-56, 1979), also deals with the specification and code generation problem.

The oil discovery system covered briefly in the box above is discussed at more length in *Case Study Report #1: Oil Discovery Problem*, by Margaret Hamilton and Ron Hackler, published by Hamilton Technologies Inc. in February 1986.

Some software experts disagree

Some industry experts disagreed strongly with the premises of this article, saying that it is not possible to build defect-free software today. They contend that the power of high-order design languages to provide reliable software implementations from specifications has been exaggerated.

Although management discipline and proper use of high-order design tools will generally increase the reliability and efficiency of software production, said one critic, there is no evidence that software tools for verifying formal specifications and turning them into executable code will help in this process.

The article's statement that proof of correctness techniques—which attempt to verify an algorithm's performance mathematically—has been unsuccessful except for very small algorithms was seen as undermining claims that formal specifications and software tools can produce logically complete and consistent software.

"Recent software engineering publications," said another industry expert, make it clear that it is not possible to develop tools for detecting logical redundancy, inconsistency, or incompleteness of specifications and that "zero-defect software" is impossible to build today.

Another critical comment was that "defect-free software" is a slippery concept: "My experience with software used to solve numerical problems has convinced me that even when the software reliably follows the specifications of the user and has been checked out on a number of test cases, there always seem to be other cases for which numerical conditioning can obviate the usefulness of the software.

"This paper deemphasizes the enormous complexities in the development of very large-scale software systems. It argues that once we get into a difficult problem, we often find that things aren't as bad as they seemed from the outside. While this is often true, it is not always true. I doubt that even with infinite time one could develop a very large-scale software system that was error-free." —*Ed.*

About the author

Margaret Hamilton (A) is president of Hamilton Technologies Inc. From 1976 to 1986 she was president and chairman of Higher Order Software Inc. of Cambridge, Mass., a company founded to develop methods for producing reliable software systems. Before that, she worked at the Charles Stark Draper Laboratory in Cambridge, where she managed the on-board flight software for the Apollo program. She received a B.A. in mathematics from Earlham College, Richmond, Ind., in 1958. ◆

SPACE TECHNOLOGY

SEE P.60

The Apollo program has been cited by leaders of Japan's fifth-generation computer project as a perfect model to illustrate how a focused technological project can yield broad benefits far beyond its own mission accomplishments. In fact, it can be argued that the U.S. space program spurred technological advancements in every nation on the globe. Certain advances were necessary to get missions off the ground, so to speak, but those improvements also enabled more sophisticated missions later on. At the same time, these and other triumphs improved the way technology was applied in the commercial sector.

The case studies and chart that follow are not intended as all-inclusive of their respective disciplines, but highlight selected work and some of the fallout over the last 25 years. They are largely U.S.-based, but similar examples exist in the Soviet program. As other nations become more involved in space, their societies, too, will benefit from the fallout of technology from their space research and development. How this fallout has benefited the technical community in the United States is examined in the first article in this section.

Key to the Apollo mission, the most extensive project of the U.S. space program, was the development of a guidance and navigation computer, which enabled astronauts Aldrin, Armstrong, and Collins to land on the moon. As explained in the second article, Apollo set new standards of reliability and accelerated developments in integrated circuits.

However, there is more to space than Apollo. The Landsat series improved technology for land, ocean, and atmospheric remote sensing. Landsat has had an international effect, too, motivating the French to develop SPOT, which will become the highest-resolution imaging system ever flown for civilian purposes when it is launched in 1985.

In each of several other areas of space technology, one event stands out as most noteworthy. In telecommunications, it was the successful launch and operation of the Syncom satellites in 1963. That system demonstrated the technical feasibility of geosynchronous communications and became a prototype for the Intelsat system. In solar-power supplies, a crucial component for most space missions, Skylab is seen as the outstanding model, supporting the largest and most complex photovoltaic array ever flown. Indeed, photovoltaic work throughout the world has been led by developments in space missions. In fireproofing spacecraft, one area of materials science, radical changes were made after a fire killed three Apollo astronauts. Another area of aerospace technology, displays and controls, has advanced due to technology transfer between designers of airplane cockpits and spacecraft crew stations.

Finally, the benefits of the world's space programs extend beyond specific technologies to the management of large, complex, and integrated engineering efforts. Space programs have helped advance the status of systems engineering and the "systems outlook" more than any large technology-driven program yet undertaken.

These points are underscored in the following articles.

APOLLO: THE DRIVER AND THE DRIVEN

The sheer scale of the United States endeavor to land a man on the moon spurred a sweeping drive of technologies that have since fluorished independently. And the technological fallout from 25 years in space has affected the way we develop and manage technology and integrate systems of hardware, software, and people.

The National Aeronautics and Space Administration anticipated the depth of the impact early in its program to explore space, giving a grant to the American Academy of Arts and Sciences to study the massive technological enterprise of the space effort for its effects on technological diffusion, community structure, manpower, and society. The first volume of the study appeared in 1965. At the time Raymond Bauer of the Harvard Business School, who headed the active working group of the academy's Committee on Space, observed that "the major diffusion of technological innovation is likely to take place after its period of maximum development when surplus technologists are freed to work in other sectors."

That may be where we stand now, at NASA's quarter-century mark, though the diffusion point was reached sooner than expected. Space programs have come and gone so quickly that they have made many engineers and other technologists available earlier to other areas of application. Today many of the architects and engineers of the space program occupy pivotal positions in industry and academia. They have transferred not only technical ideas, but also managerial innovation, especially in the area of integrating large-scale systems. When NASA geared up for the all-out Apollo effort, perhaps the most ambitious total system project ever attempted by a civilian team, it attracted the best and the brightest to its in-house work force and to its contractors. The Apollo program was an astonishingly complex combination of expedient engineering and hundreds of inventions yet to be made. All were tied to a rigorous timetable involving an enormous variety of contractors and subcontractors who were drawn from almost every technical field and discipline.

Evan Herbert Contributing Editor

0018-9235/83/0900-0056$1.00 1983 IEEE

Because the Apollo program was tied to a clear national, albeit political, goal—beating the Soviets to the moon—NASA had several powerful factors in its favor. First, it was infused with almost limitless money, enabling it to enlarge its work force and to embark upon multiple approaches in solving problems. Up-front money also made it possible to support otherwise uneconomical development or manufacturing costs. For example, getting the high-reliability components for the Apollo guidance and navigation computer did not require major process innovation by the semiconductor industry; but the strict specifications for circuit reliability forced the setting up of multiple production lines and expensive quality-assurance programs.

Space-technology milestones

Spacecraft or satellite	Launch	Milestones
COMPUTERS		
Gemini 3	March 23, 1965	First on-board digital computer control. First glass delay-line registers and core main memory.
Gemini 8	March 18, 1966	First use of auxiliary storage in flight—a redundant three-track magnetic-tape unit read through voter circuits.
Apollo 4	Nov. 9, 1967	Saturn V launch-vehicle computer uses triple-redundancy and voter circuits to implement fault-tolerant approach to reliability.
Apollo 8	Dec. 21, 1968	On-board computer controls entire manned mission to circumnavigate moon.
STS-1[1]	April 12, 1981	Redundant five-computer, dual-tape memory system provides both aircraft and spacecraft control with fly-by-wire characteristics. First multiplexed data buses.
TELECOMMUNICATIONS		
Echo 1	Aug. 12, 1960	First passive communications satellite—a mylar balloon 30 meters in diameter.
Courier 1B	Oct. 4, 1960	First active-repeater orbited. Small signal loss but limited contact time. Ground stations had to track satellite.
Telstar 1	July 10, 1962	Demonstrated feasibility of wideband transoceanic communications via satellite.
Syncom 2	July 26, 1963	First communications satellite in synchronous orbit. No tracking needed and 24-hour access became available.
Earlybird	April 6, 1965	First Intelsat satellite. First commercial communications made on June 8, 1965.
Communications Technology Satellite	Jan. 17, 1976	Pioneered use of Ku-band (12 to 18 GHz) in broadcast satellites. Extremely high antenna gain resulted in an effective isotropic-radiated power of 60 dBW, allowing use of earth receivers 40 cm in diameter. Prototype for direct-broadcast satellites. Joint project between Canada and the United States.
Marisat 1	Feb. 19, 1976	First maritime communications satellite. Signals could be received from and sent to moving ground stations on ships.
Direct Broadcast Satellite	1984	First craft designed as a DBS will enable first entirely commercial communications satellite system. It will cover the entire United States.
Advance Communications Technology Satellite	1988	NASA satellite will receive and broadcast at 30 and 20 GHZ, respectively. Will have 18 fixed antennas, 6 multiple-spot beams, and on-board data processing and storage. Should increase capacity of present satellites by 100 times.
REMOTE SENSING		
Tiros 1	April 1, 1960	First satellite of NASA's earth-observation program took 22 952 surface photos.
Landsat 1	July 23, 1972	First earth-remote-sensing satellite. Spatial resolution was 80 meters.
SMS B[2]	Feb. 6, 1975	First remote-sensing satellite in geosynchonous orbit and first one stabilized in three axes.
Seasat	June 26, 1978	First oceanographic remote-sensing satellite. Spaceborne synthetic-aperture radar proved useful for geological studies.
STS-2[1]	Nov. 12, 1981	Synthetic-aperture, or "imaging" radar, flown as a payload for the first time.
SPOT[3]	January 1985	Spaceborne sensors with the highest spatial resolution for nonmilitary uses is to be flown by France with Swedish and Belgian participation. Spatial resolution of 10 meters in black and white mode is expected.
SOLAR ARRAYS		
Vanguard 1	March 17, 1958	First photovoltaic-powered satellite.
Spaceflight 71-2	Oct. 3, 1972	First flexible-substrate photovoltaic array flown (by U.S. Air Force). Allowed large surface area to be rolled up into small stowage space. Rated at 1 kW.
Skylab	May 14, 1973	The largest photovoltaic power supply ever flown—21 kW. Power-to-mass ratio was 7 W/kg.
Hermes	Jan. 17, 1976	First photovoltaic cell array (1.3 kW) on a flexible substrate that folded like an accordian, decreasing weight as well as space. Flown by the European Space Agency.
Olympus platform	1986	First generic solar-powered platform. Multikilowatt flexible array (4 kW) will be flown by the European Space Agency and will carry communications satellites. Power-to-mass ratio will be 32 W/kg.

[1]Space Transportation System (space shuttle) [2]Synchronous Meteorological Satellite [3]Système Probatoire d'Observation de la Terre

A second factor aiding NASA was its firm, powerful control of the systems engineering process. Each of the development centers had a systems engineering division whose activities were integrated by a headquarters systems office. Moreover all NASA systems offices received additional technical support and advice from contract systems-engineering teams from companies like General Electric Co., Boeing Aircraft Co., and Bellcom (formed by AT&T to aid NASA). On questionable approaches it was possible to take multiple paths and to develop contingency plans for each risk.

A third factor in NASA's favor was the constant feedback to the Apollo systems-design engineers from astronauts and ground-support operators. This feedback had a major effect on software development and integration, which was recognized as so important to systems engineering that it was overseen from a single management viewpoint, that of the director of flight operations at the Johnson Space Center in Houston, Texas.

These three factors made it possible for major contractors to work together effectively, because interfaces were defined rigorously, with great depth of detail. The issuance of interface control documents compelled early resolution of any conflicts.

How well did it all work in practice? According to George M. Low, now president of Rensselaer Polytechnic Institute in Troy, N.Y., and director of the manned Spacecraft Center at Houston's Johnson Space Center during the Apollo program, "Perhaps the most important lesson of Apollo was deliberate design to minimize complex interfaces, thus making the systems-engineering task manageable."

The enterprise also holds lessons for conducting the business of technology. One aerospace industry veteran, James E. Ashton, vice president and general manager of the Tulsa division of the Rockwell International Corp., said, "The success of Apollo proved that even very large, terribly complicated goals can be achieved by breaking problems down into smaller problems and smaller goals. These then can be fenced in and treated as entities both from a business point of view, with contracts, and from a technical point of view, with interface-control drawings. Later the success of the shuttle orbiter proved the value of that concept to the design and apportionment of the system among many contractors."

Though the practice of engineering was driven to new heights by the space program, there were other effects that now profoundly influence many aspects of life. Robert Seamans, once NASA's deputy administrator and general manager and now the Massachusetts Institute of Technology's dean of engineering and head of its Energy Laboratory, credited James E. Webb, NASA's first administrator, for vision that went beyond his role in office. He foresaw the scientific aspects of being in space, the projected spinoffs into aeronautics, and the role that university research programs could play in ensuring success.

Mr. Webb so impressed President John F. Kennedy with this vision that he was able to get an unusual item in his budget: NASA support for several thousand doctoral candidates per year. He also got approval for a NASA grants program to put laboratories on campuses. This went beyond the need for specialized Government facilities, for Mr. Webb stated that he wanted research done "not only where the best minds would be involved, but also where the work would be tied closely to the educational process."

The NASA administrator asked the presidents of the 15 to 20 universities that received those grants to sign a letter saying they would make their best effort to set up a multidisciplinary activity on campus to look at all the ramifications of the space program. Not all the university programs germinated by NASA's grants program under Mr. Webb still have such clear-cut labels as MIT's Science, Technology, and Society Program, but multidisciplinary research flourishes.

Mr. Webb was also instrumental in the positioning of new Government laboratories. He insisted that they be established on or adjacent to campuses rather than in outlying industrial parks, so there would be "a fluxing of ideas with universities." Consequently a NASA Electronics Research Center was put right in the middle of Cambridge, Mass., within a stone's throw of four large universities, rather than in the high-technology–industry concentration along Route 128.

In a similar vein the unique concentration of scientists and engineers at space centers and test sites around the United States may have polarized the subsequent location of high-technology industry there. Inevitably such a concentration of talent upgraded education in surrounding communities, which in turn contributed new local talent to attract high-technology industries. ◆

COMPUTERS: LEARNING FROM DINOSAURS

Of all the electronic legacies attributed to space-age advances, computers share popular credit with communications satellites. However, history shows that fallout from space-flight efforts has led only indirectly to the $40 computers in discount stores today.

Pushing the state of the art was not what the National Aeronautics and Space Administration set out to do. Despite the ambitious, elegant language of the National Aeronautics and Space Act, NASA's early years were affected by the overtones of a race for the international political prestige to be garnered from manned space flight. When that objective was escalated to the unprecedented technological achievement of putting the first men on the moon and safely returning them to earth, the space agency sought proven—or at least provable—technology.

Despite ongoing advances in the science of computing, NASA elected to build and fly what might become computer dinosaurs. This convervatism turned out to be appropriate in terms of mission successes and the safety of the astronauts. But even the creation of dinosaurs affected the future of computing.

Valuable lessons were learned in designing computers to fit into small, tight spaces. Advances in software development and debugging emerged, including a new method for program interrupts that today is called memory-cycle stealing. And, most critical for the space effort, designers and vendors learned new approaches to component reliability.

As the Apollo moon shot effort moved into high gear in 1961, not all the technical problems had been solved. Eldon C. Hall of the Massachusetts Institute of Technology's Instrumentation Laboratory in Cambridge, who led the development of the Apollo guidance computer, reminisced that if his design team had known then what they learned later, they probably would have concluded they could not build the computer that was needed with the technology of the early 1960s.

As the manned space-flight program took shape, there was little question that the computer power then available could provide real-time guidance and control for the spacecraft. The tech-

Evan Herbert Contributing Editor

nology had been proved in most of the ground-based computers in existence in 1960. These were parallel general-purpose computers that had been designed in the late 1950s. The trouble was that few such machines, though they had the power for the contemplated moon missions, had been designed for the aerospace environment. Those that had been squeezed down to meet the size, weight, and power-supply parameters also embodied substantial compromises in computational performance.

The moon mission was extremely complex. It required enormous computing power that would be reliable and self-sufficient, for the spacecraft would be in a communications "shadow" during part of its translunar flight. Functioning as a general-purpose computer, the spacecraft's computer system would have to solve the guidance and navigation equations for the mission. As a control computer, it would handle such major functions as aligning the inertial measurement unit, processing radar data, managing the astronauts' displays and controls, and generating commands to control the engines.

On top of that, two computers were needed for the Apollo missions—one in a command module and the other in a lunar excursion module. This particular configuration of the Apollo spacecraft was the culmination of a technical controversy over whether the guidance problem for lunar orbit rendezvous could be solved at all.

Fortunately, rendezvous technology had been advanced considerably by military projects aimed at satellite interception. If it was possible to maneuver an unmanned inspection vehicle alongside an orbiting satellite, why not guide and dock together two manned spacecraft with essentially the same techniques? The guru of guidance and navigation, Charles Stark Draper of MIT's Instrumentation Laboratory, was so certain of the technological feasibility that he volunteered to go along on the first mission to run the computer.

The MIT laboratory had been working on basic guidance and navigation problems since the late 1950s under an Air Force study contract. From this work, on flight-path analysis for a spacecraft to photograph Mars and then return, emerged a unique design for a guidance and navigation computer. Though the Mars computer was never built, some of its features would be carried over into the Apollo guidance computer, as would the packaging techniques for a guidance computer developed by the MIT Instrumentation Laboratory for the Navy's Polaris submarine-launched ballistic missile.

By August 1961, MIT had received the NASA contract to design the Apollo guidance computer. The engineers of the Instrumentation Laboratory, which today is the Charles Stark Draper Laboratory Inc., were faced immediately with the constraints of top-down design. Within a week of President John F. Kennedy's proclamation that the United States would put a man on the moon, a reasonably well-defined Apollo spacecraft had been unveiled. The computer engineers were told they had a small hole in the equipment bay to fill with a functional system. It was to guide and control the entire flight to the moon and back, including rendezvous and linkup with the lunar excursion module. The area in the command module available for this task consisted of 1 cubic foot for the computer and some panel space for two display and keyboard devices for the astronauts [see photo].

To pack the system into that space would require high-density information storage. A suitable read-only fixed memory had been developed for the Mars computer. It was a core-rope memory, a transformer type that depends for its storage on weaving patterns into sensing wires at the time of manufacture. A stored bit is a 1 wherever a sense wire threads a core, and a bit is a 0 wherever a sense wire is not threaded through a core.

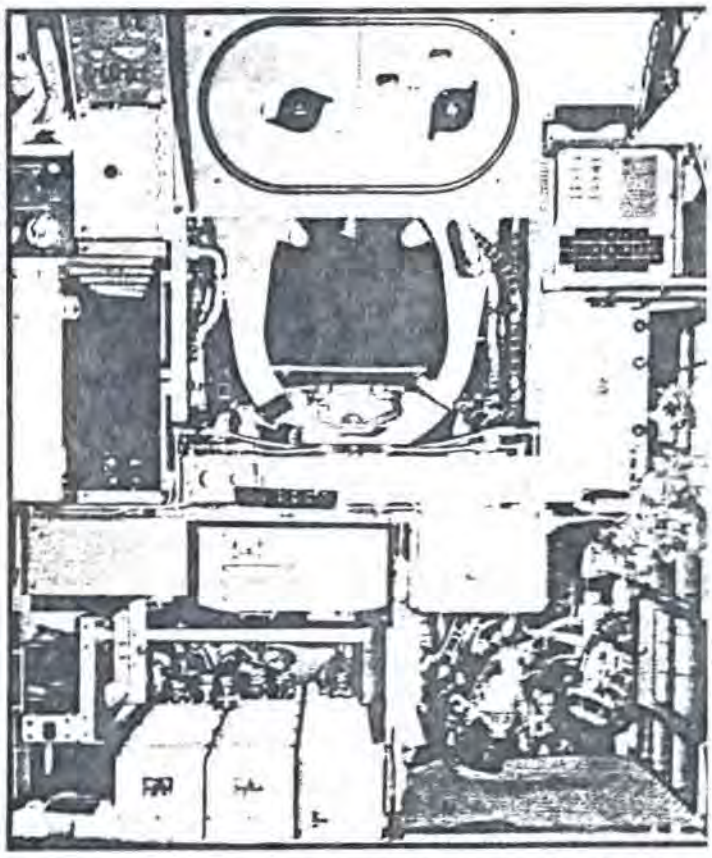
Though the content of a core-rope memory becomes unalterable, the medium is highly reliable, offering a density of 1500 bits per cubic inch, including all driving and sensing electronics, as well as interconnections and packaging hardware. At the time this was a 5-to-1 improvement over currently available coincident memory. There was room for six 6-kilobyte rope modules in the computer, offering 36 000 16-bit words of fixed storage, which would require advanced delivery of thoroughly debugged software so the information could be wired in.

The additional lead time imposed by hardwiring may have been responsible for the extraordinarily error-free mission performance of Apollo computers. It bought more time for checking and debugging after a memory had been manufactured for a particular mission.

Still, there was room for inadvertent error, as former Astronaut Jim Lovell recalls. He was the navigator on Apollo 8, the first flight to the moon to test the navigation capability of the system. On the return trip he accidentally punched up the wrong program, and the guidance system began to present information as if the spacecraft were still sitting on the launch pad at Cape Kennedy. It also lost all reference data to determine the attitude of Apollo 8 with relation to the celestial sphere. Mr. Lovell recovered by peering out the portholes for recognizable stars and manually updating the guidance system via the 2048-word coincident core-erasable memory to get the inertial attitude platform back into alignment.

Architecture and hardware contributions

Though the Mars computer design had provided some precedents, a significant contribution to aerospace computing archi-



Charles Stark Draper Laboratories Inc.

Apollo engineers were presented with a 1-cubic-foot hole in the command module equipment bay (outlined in black) into which they had to cram the guidance and navigation computer.

tecture was a program-interrupt method of accommodating real-time inputs and outputs. Large-scale computers had begun to use it prior to the Mars project. The interrupt method was incorporated into the Apollo guidance computer for single access to memory to accomplish incrementing or shifting. This process, now known as memory-cycle stealing, has been widely used to link computers to peripheral devices.

One of the architects of the Apollo design recalls his reason for selecting core-transistor logic: it already was a proven technology, and the transistors would be turned on only when used, thus requiring less power. This translated into less weight and space.

Meanwhile the hardware technology available to aerospace computer designers was developing rapidly. Because the Air Force had been pushing the development of integrated circuits by the semiconductor industry since the late 1950s, it seemed to the project manager, Mr. Hall, that the burgeoning computer might be squeezed into state-of-the-art circuitry. However, he was concerned about reliability. An estimate of component failure rates and component counts showed that the resulting computer failure rate was too high to ensure a successful mission. Reliability could be increased by conventional techniques of redundancy, but the design would then exceed the requirements for power, size, and weight. The alternative was to seek computer reliability through more reliable components and manufacturing procedures.

The number of component types and range of values was constrained to a select few, bringing complaints from the circuit engineers that their designs would be constricted. An iron-clad flight-processing specification oversaw the quality of component lots. Defective lots were subject to wholesale rejection, with no partial givebacks. Vendors complained bitterly, but what they were forced to do in order to achieve reliability was underwritten by the ample budgets of the Apollo program. Later they accepted the reliability lessons learned from required internal visual inspection before crates were packaged or systems were sealed and all parts were burned in. As a result, the Apollo computer was a pioneer in the flight into space of nonredundant integrated circuits.

A delayed impact of software

Because the computer programs were entered into the core-rope memories by the actual manufacturing process, software had to be developed and debugged well in advance of each flight. By the time Apollo 11 lifted off the pad, the software team, led by Margaret Hamilton, had analyzed software management and development techniques and the software itself. The team found that 73 percent of the problems were interface-related— data and timing conflicts within the software. Moreover, 60 percent of the problems found in a given program release turned up in other releases already approved. Though they expected to solve the software reliability problem by eliminating errors, it turned out that a major improvement came from defining a system in such a way that interface ambiguities could no longer exist among the software modules.

The empirical study of the Apollo software enabled the categorizing of axioms and definitions for scanning whole classes of errors—program statements that were inconsistent, redundant, or logically incomplete. This effort evolved into a theory of higher-order software that was applied to system definition processes to get rid of interface errors. Ms. Hamilton set out to prove the theory, for she felt that it could be the basis for automating not only the verification process, but also the software-development process. She is now president of a company called Higher Order Software, with fully automatic software products that generate bug-free systems.

Impacts of other computer activities

Though the manned spacecraft program made the greatest use of the new capability of advanced computers, they were also used extensively for simulation, preflight checkout, launch control, in-flight monitoring, and data reduction from telemetry. James E. Tomayko of Wichita State University in Kansas said NASA enjoyed its greatest success in computer simulations. These allowed experiments with equipment and plans, as well as extensive crew training. Mr. Tomayko saw that the perfect flight record of the Saturn booster proved the efficiency of modeling that aided the development of the launch vehicle. He credits NASA's continued refinement and use of simulations to the fact that the space shuttle was man-rated on its very first flight.

The 4Pi series hardware and software used today in space shuttles was influenced by the special-purpose flight computer that IBM Corp. built for NASA's Gemini program in the 1960s. It made the first use of on-board auxiliary storage—a magnetic tape unit with identical programs on three tracks, read through voter circuits before the data entered the computer. It was another approach to reliability—through redundancy. Multiple redundancy and fault-tolerant equipment flies on today's shuttles in systems configured of five IBM 4P8/AP-101 computers and two mass-memory units. ◆

Aerospace & DEFENSE UPDATE

SDI in Massachusetts

A look at the SDI contract work being conducted by local companies.

Staff Report

Massachusetts, particularly Cambridge, is one of the centers of research for the Strategic Defense Initiative program. Here is a sampling of some of the work being done in the Bay State by local high-tech firms:

HAMILTON TECHNOLOGIES, INC.

Margaret Hamilton, the founder of this small Cambridge firm, was in charge of developing the navigational software that guided the Apollo astronauts to the moon. Her experience in the Apollo program led her on a quest to find ways to write error-free software. Her first company, Higher Order Software, pioneered the computer-aided software engineering industry. She left Higher Order to set up her new company and develop a new commercial product capable of buiding zero-defect software systems "that are capable of handling change and unpredictability." Hamilton's expertise in zero-defect software is also being applied to the SDI program — her firm has built a model of a tracking algorithm.

ITEK OPTICAL SYSTEMS

"Our work matches our title," says Richard Wollensak, Itek Optical Systems' vice president of program development. "We develop optical systems, exclusively for defense programs." The Lexington-based division of Litton Industries is a "sizable contractor to SDI," he says.

Itek participates in at least two major SDI efforts — the "SATKA" (Surveillance Acquisition Track and Kill Assessment) and "DEW" (Directed Energy Weapons) programs. For SATKA, Itek is helping to develop sensors that can detect missile launchings and determine their final course. Itek's work in the DEW program includes research into both ground and space-based high-energy lasers, as well as particle beam and other directed-energy weapons.

Itek has expertise in both the optical and laser sides of DEW research. Itek uses laser interferometers to build incredibly precise optical and mirror surfaces. "We are building mirrors that are four meters in diameter," said Wollensak. "The precision of those surfaces is 1/50 wave RMS, or 1/10 of a wavelength of light. If that four-meter mirror were enlarged and spread from Boston to Chicago, that tolerance equates to no hill or valley more than three inches between the two cities." DEW needs large mirrors like that, he added.

The Litton division is also working on "active mirrors" for the DEW program. "These are thin face plates or sheets of glass," he explained. "Bonded to the back of this thin sheet are hundreds of actuators which can be commanded to pull or push on the glass and bend it. They are sometimes called rubber mirrors. On command, we can change the shape of the mirror, up to thousands of cycles per second." When connected to sensors that sense turbulence and other disturbances in the atmosphere, Itek's researchers expect, these actuators will be able to bend the mirror in a way that will allow laser weapons to compensate for atmospheric turbulence, and reach their targets with sufficient power to destroy them.

KAMAN CORPORATION

Kaman Corp. of Bloomfield, Conn. is an approximately $600 million corporation that has a secure spot in the defense industry pantheon as a supplier of helicopters to the Navy. Its Cambridge-based Electro Magnetic Launch subsidiary is conducting work in what its president, Fred Smith, calls wavefront control — taking the aberrations out of a directed energy beam or laser, so that it can find, identify, and engage targets.

LINCOLN LABS

MIT's Lincoln Laboratories is the largest defense R&D contractor in Massachusetts, and SDI-related programs are now at the center of its work. The Cambridge-based not-for-profit organization has long done work in surveilling and tracking ballistic missiles, and discriminating between decoys and weapons; it is continuing to do so under the SDI program, according to its director, Walter

(continued on page 26)

The Times Are A'Changing For the Defense Industry

Firms are adapting to changes in funding and procurement policies.

by Allan E. Alter

The 1980s have been good to the defense industry, thanks to the Reagan Administration's military buildup. But now, say industry observers and executives, the days of relatively easy money are over.

"The defense industry is becoming more difficult as a business," says Michael Marx, the vice president of marketing at Textron's Avco Systems division in Wilmington. "The reasons are tight budgets, a worsening deficit situation, and the Congressional mood. The Reagan defense buildup has come to an end, and we are looking ahead to several years of tight budgets."

> "The defense industry is becoming more difficult as a business." —Marx

The electronics, computer, and software industries will be less affected by changes in Pentagon spending than others, according to Paul Bedard, the associate editor of *Defense Week* in Washington, DC. "It will be harder for tank and truck makers to get new contracts out of the Pentagon, because the money isn't there." Instead of building new missiles, aircraft, and ships, he predicts, the Defense Department (DOD) is likely to overhaul them, "and put in new electronic guts." The new procurement rules, he added, will probably affect high tech companies less than firms in other industries. "Most of the reforms have been written to get back at the crooks in the defense business. A lot of the new rules have hit heavy industries, like aircraft makers."

However, high-tech contractors are hardly immune to these changes. Philip Phalon, senior vice president, corporate marketing, Raytheon, expects to see a tighter defense budget in Fiscal 1988. "We'll be lucky to maintain three percent growth," says Robert Bowes, a principal contracting official at the Air Force's Electronics Systems Division at Hanscom Air Force Base. "If it turns out we don't maintain three percent real growth, you are dealing with a shrinking budget." Either way, says Bowes, "that means as programs mature and new ones come into existence, there will be a tougher fight for the dollars available. We will see programs fight among each other. We already are seeing program consolidation among the armed services, and as the budget tightens we will see more emphasis on that."

"We have increasingly become more concerned about being cost competitive," says Marx. "All around Route 128, you are seeing efforts by defense contractors to significantly reduce the cost of business by reorganizing, reducing work forces, consolidating, etc. All of us are feeling we have to be able to do more work for fewer dollars."

Marx sees many other implications of these budget changes. "Given that environment, the industry is apt to become more competitive, because there will be fewer dollars to begin new projects as well as to support existing commitments. We will find ourselves in an industry where it will simply be more difficult to obtain new contracts and keep existing programs sold."

Changes in the rules governing R&D funding have also changed the defense business, says Marx. The DOD is now demanding fixed cost contracts for R&D work. "In essence, they expect the contractor to assume the kind of risks that the government was historically willing to assume," says Marx. "In R&D programs, it is uncertain how much can be accomplished and how much it will cost. Doing R&D work at a fixed price is very risky." The Pentagon has also broken away from another "historical pattern:" companies involved in early stage R&D contract work "were reasonably assured of finding themselves in production, where they could recover their investment. Now competition is being introduced in every phase of a product's evolution. You can bring a program to a certain stage of maturity, and find yourself in competition at every stage of the program." In other words, says Marx, the rules have changed — "the old system of making early investments with some assurance of long-term payback" is gone. "You still have to make early investments, but the assurance that there will be a payback in later stages is far less."

> "We'll be lucky to maintain three percent growth." —Bowes

Contract delays are another problem for contractors, says Bedard. "The services keep promising that contracts will be announced, and then delay them a year or two. Some companies blow millions of dollars trying to get contracts — they can't wait that long."

"What all this boils down to," concludes Marx, "is that the lower levels of funding, the program stretch-outs that have been resulting, and the shifting of risk, all make the profitability for the defense contractor much less reliable. So this environment is becoming, from a business standpoint, more difficult, and one might say less attractive."

THE VIEW FROM HANSCOM

Bowes acknowledges that the regulatory environment is unstable. "The regulations are constantly changing. Congress continues to be very active in procurement legislation. All of us in the DOD have said that perhaps it's time for Congress to slow the legislation down — let's start working with what they've given us. Everyone needs a more stable environment."

(continued on page 27)

SDI

(continued from page 25)

Morrow. Lincoln Labs is also conducting research in the propagation of high-energy laser beams.

RAYTHEON

"Raytheon is not a large SDI contractor," says Philip Phalon, the senior vice president of corporate marketing at Raytheon. "In the fiscal year 1986, we were number 26 in the number of SDI contracts awarded. Most of the programs which we are involved in that are today considered SDI programs existed prior to the existence of SDI, but were swept under the SDI umbrella."

Raytheon's contribution to SDI consists of a $175 million contract to develop a prototype "terminal imaging radar" (TIR) system. The word "terminal" refers to the final phase of the SDI scenario — defense against incoming nuclear missiles which have managed to reenter the Earth's atmosphere. The funds will be used to construct a phased array radar system which will be used to demonstrate and experiment with new technologies for tracking incoming intercontinental ballistic missiles. These technologies, it is hoped, will enable the U.S. to discriminate between nuclear warheads and decoy warheads.

SATCON TECHNOLOGY

The Strategic Defense Initiative Office is just one of many federal agencies — including the Air Force, Navy, NASA, Department of Energy, and National Science Foundation — that is funding SatCon Technology Corp. of Cambridge.

SatCon was spun off of MIT and Draper Labs for the purpose of developing magnetic devices that can suspend objects in space, and thus stabilize, isolate, and control them to a much finer degree than was possible until now. Magnetic suspensions could create vibration-free environments; magnetic pointing mounts could be used for precise targeting and photography; and magnetic bearings could be used in optical computer memory devices and buffers. One of SatCon's projects for NASA is the development of a magnetic control system to support a 1.2 million rpm rotating mirror system; others include flywheeel energy storage systems and momentum exchange altitude control systems. The firm is now conducting research into magnetic bearings that utilize superconductivity. □

Chubb

(continued from page 24)

game. And, by the way, there are a lot of people in that business in software and in electronics, so it's ideal for us, and we believe in it. So that's a place where the laws have been good, where the bureaucracy has made it happen and if you didn't have the strength of the bureaucracy it would have never happened, frankly. And there's all the business available for those folks that they want.

Now, there are two sides to that story. When they get that business they've got to perform, and usually they do, sometimes they don't.

MHT: And you personally, what kind of relationship do you have with the civilian leaders in the technology — do you make contact very often with your counterparts over at Raytheon and other firms?. . .

CHUBB: About once an hour.

MHT: So it's a pretty close working relationship?

CHUBB: We've done a couple of things. First, we run a pretty intense operation around here with the contractors. That's one thing that's caused a lot of contact, the other is we started a council (the Military Affairs Council) two years ago as an experiment, and this was mostly to make contact with the non-defense people.

Generally those things work well in small towns where there's a base and the town's very dependent on the base, but generally may or may not work well around large cities like Boston. However, here, the president of BayBank, Giles Mosher, agreed to take on the chairmanship of it, and that brought with it a lot of non-defense people and brought with it getting the word out to the non-defense world and to small business. That (the MAC) still relates (to them) that they can compete on things that they didn't know about. That's been a real success story here in the last two years, far beyond anything we ever believed. A lot of the credit goes to Giles Mosher and the team that he's got working for him. People in this area didn't know much about Hanscom or ESD, but now they do.

You're seeing a great deal of controversy over how much defense really means to the state of Massachusetts, and some people say it doesn't mean anything. But, in fact, Massachusetts is fourth in the nation in defense dollars right now. We're the third largest "business" in the state. We've got 29,000 cars on this base, just to give you an idea of small business interest on this base in another sector.

We probably create in the Middlesex County area 50-70,000 jobs. . .worth $1-2 billion depending on how you calculate it. And you can take the combination of the defense dollars in this state and the high tech work, particularly on 128, and the education base with schools like MIT and 30 other universities around here, and you've really got an exciting kind of business opportunity. Most of us believe that's why you don't have any unemployment out in this area — it's that factor of the three of them going together. You pull out any one of them and it won't happen.

Now we also believe that our own business, electronics, would tend to stay in Command and Control because you have to have it, particularly if there's a scale-down in nuclear missiles. It becomes even more critical. We believe that our budget will tend to stay more or less the same through administrations whereas some other areas, where you're developing brand new weapons, the systems may go up and down. So we see a continuing good economic situation in this area because of the defense dollar, high tech industry, and the education base that exists in this part of the Northeast.

MHT: Will MITRE be expected to grow much?

CHUBB: No, maybe a little, but not much. The reason for that is, and I think it's correct, that Congress caps the Department of Defense at certain levels, both the military and civilian sides. So regardless of how large our business gets, our numbers of people will stay roughly the same. And you will say, 'how can you handle that problem?' Well, you shift more of the work to the contractor, which has its pros and cons. Its "con" is that it's hard for him to really focus on your objectives as well as you can. . .

MHT: How closely do you end up managing MITRE?

CHUBB: We work very closely with them.

MHT: But manage per se? Do you tell "Caz" Zraket [MITRE President] what to do?

CHUBB: Well, manage is the wrong term. We've got an organization called Electronics Systems Division. MITRE's job is to provide system engineering support to ESD. Now that doesn't mean I go manage Caz Zraket. That means that his organization has to provide to each of our programs, AWACS, JSTARS, SDI, technical support for each of those people working on those prog[illegible]. That's a very direct and a ver[illegible] relationship. But he is actual[illegible]ndent company, but char[illegible] work for us up here. □

CASE: The New Force in High-Tech

by Allan E. Alter

With the possible exception of the Central Artery, the worst bottleneck in the computer industry today is software development and maintenance.

The full power of many computers, ranging from parallel processors to PC ATs, remains untapped because software engineers have yet to create systems to run on them. Software bugs are as persistent a problem as cockroaches. These electronic vermin not only cost end-users and software companies millions of dollars and man-millenia of effort every year — they breed lawsuits as well. And companies often wait months, if not years, for their DP or MIS departments to create applications and systems. "Backlog" is now a full member of the pantheon of computer buzzwords.

Nasty stories abound on the time, money, customers, and opportunities that have been lost because of software problems: the financial services company that ground to a halt because a bond-tracking program with room for only 32,767 bond issues had to process a 32,768th; the time when a radar station in Greenland mistook the moon for a massive missile attack; the bug that cost the Bank of New York $5 million; the glitch that caused ITT to abandon a new product.

But don't feel sorry only for the end-user — consider the plight of software developers. They spend endless hours of boredom writing applications code line by line. Since there is no systematic way to reuse code that is known to be bug-free, software writers often repeat work that has already been done. Furthermore, most people who design software systems are reduced to pencil, paper, and templates to sketch out logic paths and data flows.

"We have three themes: automation, integration, and reuse." —Addelston

Since the early 1980s, software professionals have had a few allies in the battle to create accurate, bug-free code — fourth-generation languages, compilers, debuggers, and the like. But today, new tools are emerging which speed up the production and improve the quality of software. Together, they make up an emerging industry that has, in the last year or two, come to be known as "CASE" — computer-aided software engineering.

FRONT END, BACK END

The nascent CASE industry provides tools to automate the job of writing and maintaining (i.e., fixing, updating, or improving) software, much as CAD/CAM and CAE do for design engineers, drafters, and manufacturers. It's a long process with many steps — analyzing what the software is supposed to do (a task often undertaken by or with the end-user); setting precise specifications; designing the system; creating actual code; testing the software to make sure it is error-free and logically consistent, does what it is supposed to do, and runs on the hardware platform it is meant to utilize; and maintaining the whole package.

At this early stage, the CASE industry tends to divide into three groups — companies that provide "front-end" tools for designing software such as Index Technology of Cambridge, Cadre Technologies of Providence, R.I., Nastec of Southfield, Mich., KnowledgeWare of Ann Arbor, and MacDonnell-Douglas of St. Louis; firms that offer "back-end" systems for automatically generating code, including Pansophic, CCA, Sage Software, Microfocus, and Transform Logic; and a smaller circle of companies which offer both, including Cortex Corporation of Waltham and two Cambridge-based firms, Higher Order Software and Hamilton Technologies.

Several well-known high tech companies are also participating in this market. Both Cullinet and Wang Laboratories have come out with front-end CASE tools that work in their own environments — Cullinet's *IDMS/Architect* and Wang Labs' *The Solution*, developed by the Hartford Insurance Group on the Wang VS operating system. DEC and IBM are said by industry observers to be developing CASE tools of their own. However, no one company dominates this new industry.

Even at this early stage, it is clear that CASE will not only change the way computers are used — it will change the computer industry. CASE tools can save time and thus money. They can free software developers for more important tasks that call for human judgement, and speed time to market for companies that sell software as a product or as a component of a larger system. And they will help manufacturers of non-standard computers such as parallel processors by enabling them to build new applications more quickly, and to port old applications more easily. CASE is becoming an important market for workstation vendors.

Margaret Hamilton, president of Hamilton Technologies.

Ultimately, CASE will change the way work is done in MIS and R&D departments. It will create new kinds of management and systems analysis positions, and eliminate others. "If you are a guy writing COBOL applications, then I would start looking for other things to do," says one CASE company executive. "If you are a systems programmer, you are in good shape." It will also have an impact on CAD/CAM. Like CAD/CAM, CASE builds models; while some feel CASE tools could become front-end systems to CAD/CAM systems, others believe the two technologies could some day compete. Industrial companies like DuPont are already using CASE tools to model factories and processing plants.

The potential of CASE is so great that 14 of America's best known and biggest aerospace and defense companies — including Boeing, General Dynamics, Lockheed, Northrop, and TRW — have set up a joint R&D organization called the Software Productivity Consortium in Reston, Va. The mission of the Consortium's 120 staff members is to develop CASE tools for mission-critical software. The group is now creating a library of reusable "standard functions" that can be configured to particular environments, integrating CASE tools from the entire software development cycle, and integrating CASE tools with project management tools. "We have three themes," says Jonathan Addelston, vice president of the Consortium's software product development group: "automation, integration, and reuse."

MOON CHILDREN

But here in the Route 128 belt, competition rather than cooperation is still the rule, and the big names in CASE are still small companies.

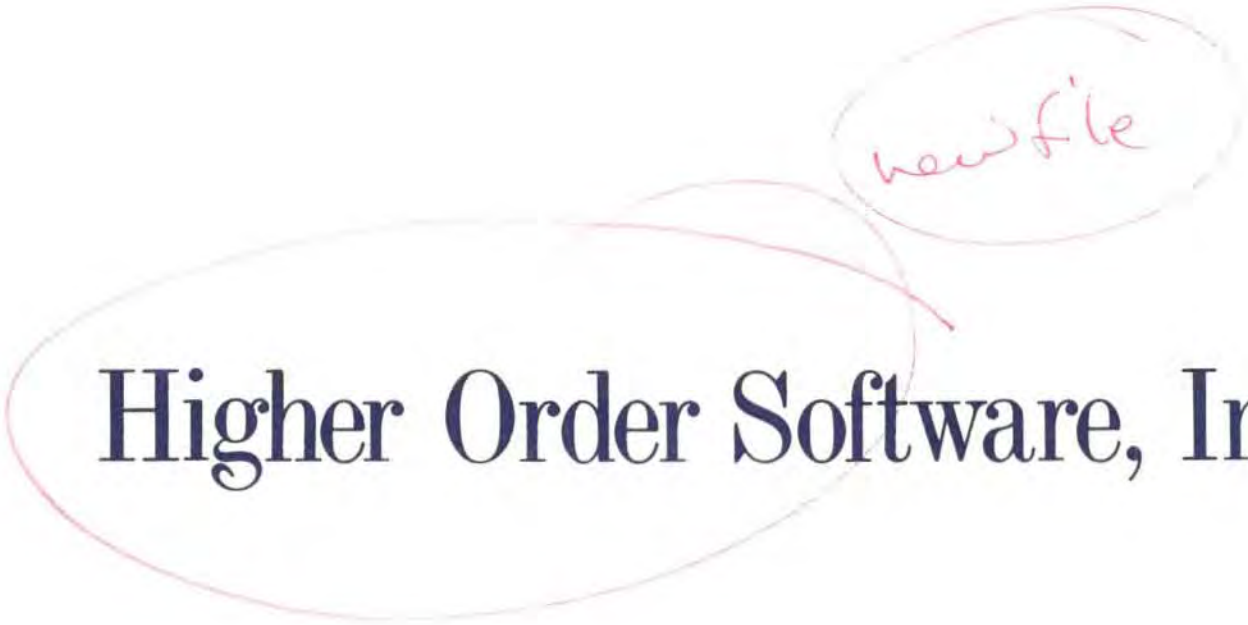
Two Cambridge-based CASE firms share the same founder — **Higher Order Software** (HOS) and **Hamilton Technologies, Inc.** (HTI). The founder in common is Margaret Hamilton, who was in charge of developing the navigational software that guided the Apollo astronauts to the moon. The common thread that unites the Apollo mission and the two companies is Hamilton's quest to write error-free software.

Higher Order, Hamilton's first venture, released its *Use.It* system in [illegible]. As Hamilton explains it, "If requirements are stated in a formal and practical way, then it is possible to automatically translate requirement statements into lower-level, machine-readable, executable forms."

Margaret Hamilton [illegible] employees of Hamilton Technologies is focusing on the high end of CASE. While Hamilton subscribes to the same philosophies that led to the development of Use.It, she is developing a new technology, called *001*, for building zero-defect software systems "which are understandable, capitalize on reusability," and are capable of handling "change and the unpredictable." "We are definitely a candidate for supercomputers," she says.

But rather than provide off-the-shelf software, Hamilton is now building large systems for her clients. She is using 001 to create a model of an entire factory — "hardware, software and peopleware" — for one company. The other client is the Defense Department's SDI program....

1984



Higher Order Software, Inc.

Company Profile

TABLE OF CONTENTS

HIGHER ORDER SOFTWARE, INC.

WHERE WE CAME FROM AND WHAT WE PROVIDE . . .
THE DELIVERY OF AUTOMATED SOFTWARE ENGINEERING

Higher Order Software, Inc. is a software products and services company founded in 1976 by Margaret Hamilton and Saydean Zeldin to develop and market a revolutionary software methodology and the USE.IT products that express that methodology.

Our products and services have developed through reinforcing stages of implementation, evaluation and enhancement from that time.

In 1984, we number over one hundred employees and are eminently experienced to provide you with a complete commitment of support unequalled in the industry.

Quite simply, HOS stands alone in the delivery of automated tools for computer-aided software engineering.

Our methodology makes it possible - for the first time - to employ an automated and verifiable approach to software engineering throughout the entire life cycle.

This means that the software engineer can now use one method and one graphical, interactive tool and design in one completely interlocked process of project management and implementation. In fact, HOS and USE.IT enable you to:

- Automate the software engineering life cycle.
- Automatically verify the logical completeness and consistency of all design components.
- Automatically generate correct computer code from a verified design.

The delivery of computer-aided software engineering means tremendous productivity gains for everyone involved in a large systems project -- and corresponding impact on the bottom line . . .

OUR MARKET AND OUR CLIENTS . . .
AMERICA'S LEADING CORPORATIONS IN EVERY ARENA

Applications In Every Category

Because our methodology captures the high-level abstractions at the foundation of all software engineering, our clients can target USE.IT across many traditional application barriers.

Our market stretches from banking to aerospace, from database design to communications networks. Our clients are using HOS, to cite but a few applications, for:

- Design and build microprocessors
- Construct programmable controllers
- Simulate experiments for the space shuttle
- Build a fault tolerant operating system
- Create human resource management systems
- Interface to industry DBMS products
- Build weather, seismic and other geophysical devices
- Develop a communications network for funds transfer
- Monitor a major manufacturing facility
- Develop a financial modeling system for end users

Partnership With Distinguished Clients

Historically, our clients have spanned a wide range of commerical corporations and government agencies. The clients who use our products and services are richly represented in the Fortune 200 category.

We don't sell to these corporations in the traditional sense. Instead, we enter into a serious, sophisticated dialogue with them that has decades of future relationship in view.

Our clients must qualify us . . . and we must qualify them. Naturally, we spend months of our time to develop the trust and mutual commitment needed to conduct the professional collaboration that marks true software engineering.

A partial list of HOS clients includes:

ADT
Allen Bradley
Army - Computer Systems Command
Bank of America
Boeing Computer Services
Cincinnati Milacron
Citibank
Computervision
Control Data
Data General
Dept. of Transportation
Digital Equipment Corporation
DuPont
Eastern Kentucky University
Eaton Corporation
French Ministry - CNET
General Dynamics
General Motors - PMD
GTE
Harris Corporation
Honeywell
ICAM/Natl. Research Council of Canada
Interelec
Lockheed
Martin Marietta (Dept. of Energy)
Matra
Metra
NASA - Singer/Link
Raytheon
Research Triangle Institute
Rochester Institute of Technology
Scott Paper
Syscon
Systems Development Corporation
Tascal
Teledyne Geotech
Texas Christian University
UCCEL
U.S. Navy - FMSO
Naval Weapons Ctr. - China Lake, CA
University of Delaware
University of Michigan
University of S.W. Louisiana
Wang Institute
Westinghouse
World Computer Corporation

HIGHER ORDER SOFTWARE, INC.

OUR LEADERS . . .
A HISTORY OF EXCELLENCE IN MANAGEMENT

Excellence From The Beginning

Our co-founders, Margaret Hamilton (CEO and President) and Saydean Zeldin (CFO and Executive Vice President) have pioneered a continuous history of excellence in management:

- Excellence in the management of research.
- Excellence in the management of large software systems.
- Excellence in the creation and management of Higher Order Software, Inc. from its beginnings in 1976 until today.

Margaret Hamilton managed the Apollo moon program's on-board software system - an extraordinarily large, difficult and sensitive project.

She was fully responsible for two hundred project employees and a budget that, in today's terms, exceeded two hundred million dollars. More significantly, she bore a large share of responsibility for the safety of the Apollo crew.

Because it was so vital to prevent software errors, both Hamilton and Saydean Zeldin (who supervised one of the six project groups) were forced to pioneer techniques that might have been theoretical in an academic setting but were a matter of life and death in space.

The very success of the Apollo program led to a further round of research. This, in turn, generated the HOS methodology and the AXES language. Higher Order Software, Inc. was founded with a mandate to develop the commercial implications of these revolutionary discoveries.

Breakthroughs have been leveraged since then from the solid foundation established at the beginning. As with the Apollo project, Hamilton and Zeldin have never enshrined theory over practice. To the contrary, advances have been predicated on this principle:

As we meet customer requirements and distill lessons learned in the actual practice of software engineering, Higher Order Software, Inc. will maintain the deserved leadership role within the industry.

The Tradition Of Excellence Continues

In 1984, we are proud to announce the addition of James Frame to Higher Order Software, Inc. as Chief Operations Officer and Executive Vice President with direct responsibility for Marketing and Sales, Advanced Support and Product Engineering.

Frame brings to HOS an immense range of management experiences and skills that uniquely complements the innovative pioneering of our founders.

He was the first corporate Vice President from the Programming Division in the history of ITT. While there, he supervised 9,000 software engineers and increased their productivity within three years by 44%.

Previously, he spent twenty-two years at IBM where he was intimately involved in the development and management of a host of critical products:

- IMS database management system.
- CICS (Customer Information Control System)
- COBOL, PL/1, FORTRAN, APL and BASIC programming languages.
- BTAM, QTAM and TCAM telecommunication access methods.
- DOS/VS operating systems for the IBM 370.

Frame's career at IBM was capped by his management of the 2,000 employee Santa Teresa Laboratory in San Jose, California. Designed by him from top to bottom to further software engineering, it was the first installation of its kind in the world and is still hailed as a remarkable synthesis of architectural beauty, human ergonomics and IBM software production values.

Excellence As A Leadership Team

As the company moved from a research and development posture in its early years to the current focus on marketing and applications support, Hamilton and Zeldin have translated excellence in project management to excellence in the management of a growing business corporation.

Now, James Frame, working with them, is overseeing the transformation of HOS from a successful but small computer corporation into a major American corporation recognized worldwide as the leader in computer-aided software engineering.

Their teamwork and creative leadership ensures that Higher Order Software, Inc. will remain unsurpassed not only for software products and support, but in the pioneering of a new management structure.

It is a structure that matches the uniqueness of our methodology and our determination to remain the leader in a global marketplace. It is a structure based on individual excellence, the pooling of personnel skills across departmental boundaries and the forging of corporate consensus.

Our customers reap the benefits:

- HOS salespersons are the most technically literate in the industry.
- HOS technical personnel make themselves sensitive to marketplace needs so that research is relevant.
- Special employee positions incorporate leading-edge functions unknown within traditional organizations.

Based on the contributions of our leaders, excellence in management prevails throughout Higher Order Software, Inc . . . in many dimensions and throughout all divisions and departments.

HIGHER ORDER SOFTWARE, INC.

OUR DIVISIONS AND DEPARTMENTS . . .
WORKING FOR YOU THROUGH THE LIFE CYCLE

HOS product development does not end with the delivery of USE.IT: it begins. Product research is interwoven throughout our relationship to our clients and their projects.

Likewise, HOS support services do not begin and end with the sales cycle. Our support is designed to work in collaboration with you throughout the life cycle of each working project . . . and beyond.

We don't envision software engineering as a static set of products to be sold, but as a dynamic mix of leading edge products and expert support to be delivered in mutual relationship.

For these reasons, a traditional organizational chart, while vital for conducting internal operations, is not the best way to present HOS to our customers.

Instead, we present our work functionally.

We show our customers how they will interact with our people from the first day of contact to the development of an extremely sophisticated set of projects many years after initial installation.

At HOS, we mobilize the entire company to provide a full complement of products and services in interactive, permanent - and, of course, confidential - relationship to each of our customers. Our unswerving goal is to provide error-free systems that are robust, efficient and on time.

We measure our life cycle of customer support in decades, not months or years. From purchasing to training through special project consulting services, HOS is there with you - today and tomorrow.

FROM INITIAL CONTACT TO PURCHASE WITH HOS

Comparing Qualifications

HOS is frequently contacted by companies that have heard about our revolutionary methodology and products. We are delighted to respond with diligent care to every contact.

Our experiences do show that companies who appreciate the tremendous cost penalties of traditional life cycle development and are ready for a different approach to software engineering are the best candidates for USE.IT.

For this reason, our sales force, led by Vice President of Marketing and Sales, Tom Lutz, places an initial emphasis on customer qualification . . . . and we encourage our customers to qualify us in relation to them at the same time.

After all, you need to quantify the benefits that HOS and USE.IT can provide to your unique set of applications. Customers must determine that we can complement their available resources - just as we must determine that customer needs can be reasonably and promptly met.

A striking benefit of the HOS methodology is that you can freely employ it to implement the HOS functional life cycle or easily adapt it to suit your own project life cycle.

Presenting HOS and USE.IT

Our unique methodology and product furnishes tremendous productivity benefits to our customers, but it demands the utmost technical literacy from our sales force.

We recognize that everyone from senior data processing executives to project managers deserve a USE.IT presentation that is customized to their job function and scope of authority. HOS demonstrations are designed so that customers can initiate an interactive conversation answering their unique questions and meeting their unique needs.

Demonstrations represent only the beginning of our response to potential clients. Our sales force is at your disposal whenever questions arise. We meet frequently with you during the initial stages of the relationship.

At all points, our sales force offers the utmost in personal, professional services to all management executives involved in the decision-making process. A serious, sophisticated product demands a fully professional presentation. We want clients to know that each salesperson is fully empowered to call upon all available HOS personnel to meet special requirements and to finalize commitments.

Developing The Technical Evaluation

Experience has shown that USE.IT is quickly mastered when applied to a concrete project - whether anticipated or underway.

Our advanced support team under the leadership of Allen Razdow is brought into the sales cycle from the very beginning to work with the customer so that we can structure a real world fit between HOS, USE.IT and your project. The goal of the technical evaluation is to target a specific project to USE.IT and to develop a coherent plan for post-sale training, education and support.

This means that customers can immediately quantify the hard benefits of USE.IT in financial saving and increased productivity and measure this against strategic corporate goals.

It also means that we provide, through the pre-sale presence of advanced support, some initial training in the HOS methodology and USE.IT - without charge - even before HOS products and support services have been purchased.

When the technical evaluation is completed, clients are easily able to formulate an internal plan for the deployment of HOS support and the USE.IT product.

The sales process closes in the context of a firm relationship between two professional organizations fully committed to support one another.

Concluding Contractual Agreements

The Finance and Administration division, led by Vice President David Blohm, works with you to prepare a contract for both products and services that perfectly matches each of your requirements today and plots the desired path for tomorrow.

We make it our business to carefully specify mutual obligations so that each party is fully satisfied.

The true expertise of this team is shown in their ability to quickly pave the way for the installation and initial usage of the product in an actual application. After all, their goal - as with every other employee at HOS - is to get you up and running with USE.IT. Contractual negotiations do not lead to project down-time with HOS.

The Finance and Administration division also plays a critical in-house role in their oversight of accounting, cash management, personnel and other administrative areas.

Training Software Engineers

Anyone can sell a product. But what happens after the sale? At HOS, purchase is only the beginning. We know our success depends as much upon our delivery of training and education for software engineers as on USE.IT, the automated software engineering tool.

Your training is not an afterthought at HOS, but the centerpiece of our commitment to advance the science of software engineering.

The advanced support group is charged with the development of an entire range of training and support materials -- from workshops and seminars to technical notes, books and educational courses.

These materials have emerged from the actual experiences of customers. They express today's thinking regarding usage of the HOS methodology. We update them in response to your feedback and our own in-house evaluation.

Because these HOS personnel are as adept at hands-on application development as with training, HOS materials and courses are targeted to the vertical applications most relevant to particular sets of customers.

After all, training and education are important, but we believe our primary goal is to support your ongoing project activity.

Expanding The Range Of Application

HOS Account Managers are, of course, involved throughout the sales cycle - supporting sales personnel and meeting your special needs.

It is after purchase, however, that their special skills to help you come into focus. With the participation of both sales personnel and advanced support, account managers implement the recommendations of the technical evaluation. We ensure that your first project with USE.IT is a success.

An account manager serves as your liaison to HOS and as your advisor for the expanded usage of HOS products and services. Our users report that their ability to apply USE.IT expands dramatically project by project. From the success of your first project, we help you leverage USE.IT productivity across entire families of application projects.

Although all users report dramatic productivity gains from the beginning, the true value of USE.IT explodes in subsequent applications. The reusability of both previously verified high level specifications and the creation of modular library components has no ceiling.

The application services team specializes in advanced consulting and sophisticated support of your large systems projects. In this way, our broad range of experience with USE.IT can be yoked to your detailed knowledge of your own complex requirements.

Our goal is not to supplant customer engineers. To the contrary, we find that users soon surpass us in their application of HOS methodology to their own projects. We consider that a mark of success. However, we do stand ready at all times to come on-site or to work from HOS headquarters on any project support task where we can cycle up still further the productivity of your systems projects.

Keeping Abreast Of Advanced Concepts

In the long term, we keep customers fully informed of the growing knowledge base of software engineering skills that arise from HOS usage. Since the HOS methodology signals a new way to think about systems design, we want to place knowledge resources - as well as product - at your disposal.

This intimate partnership in communication with our customers is still another distinctive and unique element of our total support policy.

At HOS, we do not neglect the support of special research projects that link systems design research, artificial intelligence techniques and other farsighted developments taking place in the industry.

Ron Hackler (Director of Advanced Concepts) and his staff keep HOS tied to this rapidly approaching future. Our experience shows that the fruits of their special research projects usually find their way into your hands a few brief months after their completion.

Without question, the key to future HOS development of products and services rests secure in the elasticity between HOS theory and methodology and the implementation of that within USE.IT. This stretch between what can be done and what has been done constitutes our most exciting challenge. It guarantees that today's USE.IT will never become tomorrow's forgotten product.

Quite simply, we see no end at this time to the development of products that simply unfold the logical consequences of the HOS methodology.

TECHNICAL ENGINEERING:
ADVANCING THE STATE OF THE ART

The Strategy: Anticipating Tomorrow

Under the supervision of Tom Key, Vice President of Engineering, HOS is extending the industry-leading reach of USE.IT and related products to anticipate and prepare for tomorrow's software engineering marketplace.

The strategy calls for a three-pronged allocation of resources to achieve measurable gains in:

Quality: Achieving a 100% problem-free product.

Stability: Producing consistent performance under all foreseeable conditions and in all environments.

Function: Porting USE.IT to additional environments and interfacing it to useful industry tools.

The Interface With Our Customers

On the one hand, customers expect the technical engineering function to be neither seen nor heard. Customers are not a beta site for HOS development. On the other hand, our clients rightly expect engineering to be 'felt' at every level.

We accomplish this by interfacing with you through our sales and marketing force. In essence, we are the servant of the marketing force as they, in turn, serve you. You will sense our presence as we through them:

- Monitor the details of installation.
- Construct the bridge between product versions.
- Manage the USE.IT link to different hardware environments.
- Respond to requests for product enhancement.

At HOS, we consider successful engineering to be the kind that is always working for our customers - but never intruding into their solution of application problems.

HIGHER ORDER SOFTWARE, INC.

WHERE WE ARE GOING . . .
TOWARD A HIGHER ORDER OF SOFTWARE

Each HOS achievement has signaled a new era in the implementation of our methodology through our products. This, in turn, has led us to greater challenges.

We will continue to lead the way to ever higher orders of automated software engineering and design.

HOS is an innovator - as a company, for our methodology and with our products:

The Company . . .

HOS has transformed itself within three brief years from a research and development oriented company to a broadly based corporation that moves with equal ease between commercial and government applications.

Over the coming decade, we project enormous expansion in revenue. We will use our expanded resources to increase our massive investment in expert personnel and leading-edge product development.

The challenge that faces us is not growth: we are growing exponentially. Our challenge is to manage growth.

We refuse to reduce our commitment to innovation in order to meet this challenge -- but we are just as determined to yoke the stability of our corporate structure to our mobility in responding to the marketplace. HOS is ready.

We also welcome the challenge to become the single major educational force within the software engineering industry. With USE.IT, software engineering can move from a black box art to a science that can be held responsible for its performance. With HOS and USE.IT, software engineering achieves accountability.

In short, we believe that our customers deserve our commitment to become a major American corporation as well as the major industry force in software engineering. They have that commitment - in full.

The Methodology . . .

Every company, explicitly or implicitly, has a methodology.

Every product of a company expresses the logical consequences that flow from that methodology.

At HOS, we make methodology explicit. We have no hidden agendas. Our products rise or fall upon the soundness of the theories that drive them.

Far from hiding our methodology, we train our customers in it.

Our eight years of experience have demonstrated the essential soundness of the theory underlying our specification language (AXES) and the automated software engineering tool that supports that language (USE.IT). Even so, we are diligently testing and evaluating HOS methodology against every customer experience.

Ideally, HOS products should transparently incorporate the full implications of HOS methodology.

We really believe that this ideal is attainable.

The HOS methodology already encourages the production of logically error-free specifications and the automatic generation of error-free code from these specifications.

We consider the tested marketplace benefits of USE.IT in forbidding entire classes of software errors to be our best answer to thorny theoretical problems of program provability. While important research continues in this area, our satisfaction rests with the work our customers are doing.

Their groundbreaking work is the best illustration of the consequences that flow from our methodology.

The Product . . .

While most of the industry can only promise automated tools, we deliver USE.IT, the automated software engineering tool that works today. In fact, the current version of the product represents several generations of thorough development, testing and enhancement.

Because the HOS methodology is so rich, there are many future development paths for USE.IT. The unique, mathematical basis of our methodology guarantees a future path for our products that cannot be imitated by competitors.

We know that we're responsible to anticipate the needs of our users before they do. And we know we need to listen to our users every step of the way. They are the systems engineers whom we serve.

HIGHER ORDER SOFTWARE, INC.

THE LEADER IN COMPUTER-AIDED SOFTWARE ENGINEERING NOW AND FOR GENERATIONS TO COME

Excellence today . . . excellence tomorrow. That is the unstated company credo of Higher Order Software, Inc.

We are a group of people who believe excellence and productivity are not merely optional but are truly vital if American technology is to retain leadership into the next century.

The Apollo moon project and the historic achievements of our principals with that project, important as they were, are just that - history.

HOS today is involved not with history but with the future.

The HOS message is just this:

Our methodology has enabled us to produce a software engineering product second to none in the world. The revolutionary method embedded within USE.IT will keep HOS state-of-the-art -- regardless of any emerging fifth-generation tools.

More important, our methodology and product is matched by a unique company structure that mobilizes us to keep and extend our lead in the delivery of software engineering education and services.

We know that tomorrow's software engineering marketplace will include major players from around the world -- and we welcome them. We are not so arrogant as to believe that we are the last word in software engineering.

We think it's enough to be the best word in software engineering.

Investors in Higher Order Software, Inc. include:

Alex. Brown & Sons, Inc.

Cazenove & Co.

Emerging Growth Partners

Frontenac Venture Company

Greylock Management Corporation

Henry & Co.

The Hillman Fund

James Martin

Merrill Lynch Venture Capital, Inc.

Montagu Investment Management Limited

Newcastle Company Limited

Sears Investment Management Co.

J. F. Shea Co., Inc.

Venrock, Inc.

HOS
Higher Order Software, Inc.

USE.IT PRODUCT BRIEF

USE.IT — BRINGING SYSTEMS DEVELOPMENT TO A HIGHER ORDER

The world's *first automated systems engineering tool* for developing systems with a Higher Order of reliability. USE.IT stands alone as a tool which automates an engineering discipline for rapid yet *correct* systems design and implementation.

"Automated" — USE.IT employs the computer to automate the system development process, eliminating human logic and implementation errors.

"Engineering" — USE.IT uses precise scientific principles to ensure the accuracy of systems design to produce efficient, economical systems. No other product combines automation and engineering discipline for the development of reliable, unbreakable systems.

FOCUSING SYSTEMS DEVELOPMENT TOWARD PEOPLE

Because USE.IT automatically applies engineering discipline to systems design, analysis, and implementation, the system developers effectively concentrate on fulfilling requirements. End users and systems analysts work interactively at a terminal screen to construct graphical specifications in a high-level top-down tree structure. This structure, called a *control map*, is easy to read and understand as the user and analyst create a model of the system.

The model is then automatically analyzed for logical correctness and consistency to detect errors before implementation. These errors constitute the bulk of errors traditionally made during the development process. Models are corrected and reanalyzed in an iterative fashion until the specification is error-free.

The automatic code generator of USE.IT, the Resource Allocation Tool, provides the ability to prototype the system or any of its subsystems so that the user can dynamically examine performance of the verified design. The ability to perform rapid and accurate prototyping provides flexibility that has long been desired in the development process.

When the entire system design has been graphically specified and analyzed, it can either be used as input for the automatic code generator or used as a specification for manual coding.

HIGHER ORDER BENEFITS

Through automation of the HOS systems engineering discipline, USE.IT represents a powerful tool for meeting the business challenges of the coming decades. USE.IT offers:

- *clarity and accuracy* through improved communication between the end user and the DP staff.
- *reduced expense* through removal of errors at the least costly stage of the development process.
- automatic generation of *reliable program code* to match the error-free specification.
- *reduced maintenance* and consistently *up-to-date* documentation.

USE.IT FACILITIES

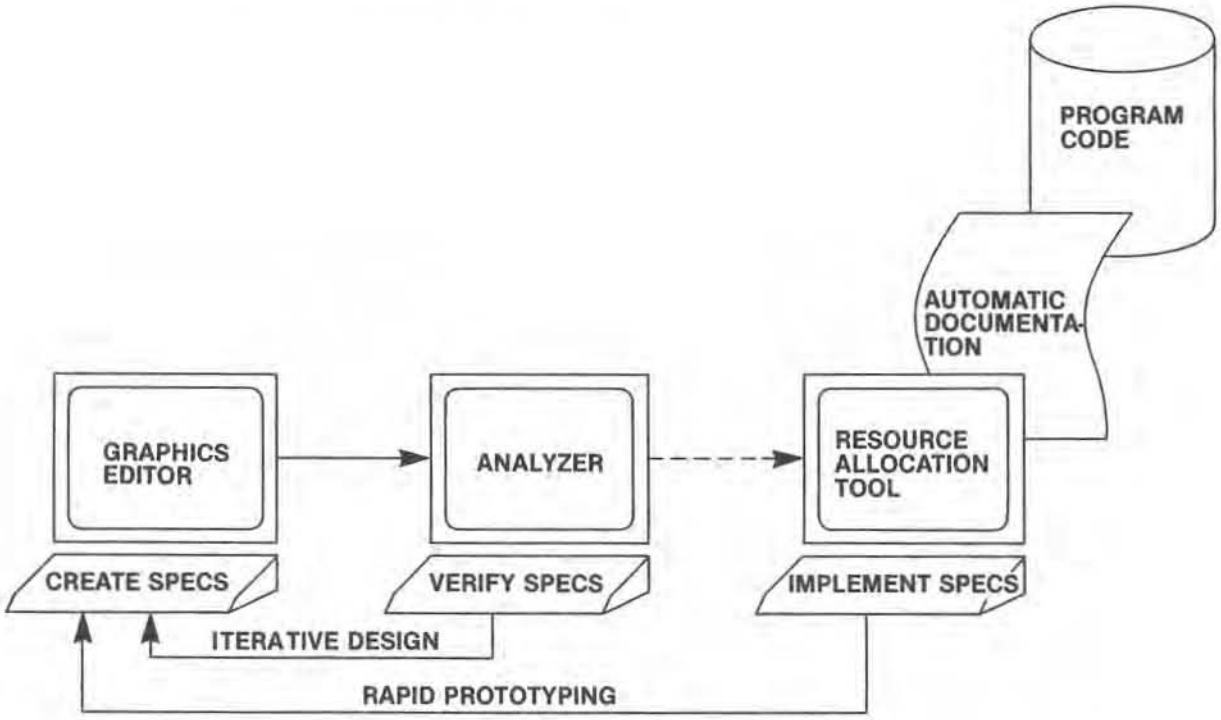
USE.IT consists of a family of integrated system development components which offer the full advantages of HOS research and breakthrough technology. The USE.IT Graphics Editor allows creation of simple yet precise specifications. The Analyzer automatically applies mathematically-based principles to confirm logical accuracy of the specifications. Together, the Graphics Editor and Analyzer allow developers to engineer system specifications in a productive, flexible way. The Resource Allocation Tool provides the means to quickly generate program code, as well as providing a test harness for rapid prototyping for the end user.

USE.IT utilities provide the means to plot graphic specifications and to produce English system documentation. In addition, USE.IT is delivered with a Library of both useful general operations and language-specific operations which serve as fundamental sets of "building blocks." A menu-driven interface allows convenient use of USE.IT and also facilitates the control, management, and tracking of system development.

2067 Massachusetts Avenue Cambridge, Massachusetts 02140 617-661-8900

SYSTEM REQUIREMENTS

USE.IT has been designed for the DEC VAX environment under the VMS operating system (3.0 and higher). Code generation is available in FORTRAN or PASCAL. USE.IT will be compatible with other computer and language environments in the near future.



LEARN MORE ABOUT THE ADVANCED TECHNOLOGY FROM HOS

For more information, including technical sales literature, regional USE.IT seminars, and demonstrations, please contact HOS at the telephone number below.

Higher Order Software, Inc.
2067 Massachusetts Avenue
Cambridge, Massachusetts 02140
617-661-8900

savant institute

U.K. OFFICE:
2 NEW STREET, CARNFORTH, LANCASHIRE, LA5 9BX
Telephone: (0524) 734505. Telex: 65138

PROGRAM DESIGN WHICH IS PROVABLY CORRECT
— *A QUANTUM LEAP IN SOFTWARE*

by James Martin

This report presents not merely a design technique but an important way of thinking about systems which ought to be understood by all DP staff.

The technique applies from the highest level of systems conceptualization down to the lowest level of program design. It results in automatic generation of program code. The automation can save much time and money.

Its main importance however is that the logic so created is provably correct. At last we have the capability to create systems which are engineered with a precise discipline to be bug-free.

In the words of the author:—
"This report describes a mathematically based approach which is really usable because the mathematics are hidden under the covers of a user-friendly set of tools. It is one of today's most important breakthroughs in the technology of designing systems.

While it was generally thought that mathematically provable software was a long way in the future, a new technique has now emerged from the work of Margaret Hamilton and Saydean Zeldin which is both highly powerful and practical. The technique has been automated so that provably correct systems can be designed by persons with no knowledge of either mathematics or programming. The software automatically generates logically guaranteed program code. Whereas mathematics like that of Dijkstra has been applied to only small programs, Hamilton and Zeldin's technique has been used successfully with highly complex systems. Furthermore the technique is used not only for program design but for high level specification of systems. The design is extended all the way from the highest level statement of system functions down to the automatic generation of code."

CONTENTS

ISBN 0 906774 28 4 **To be published November 1982** **price 200^{00}**

How to Order

An order for this report can be placed by contacting Savant Institute, Carnforth, England. Orders by air mail and telex will be accepted. Savant regrets that telephone orders cannot be accepted. Price includes delivery.

Reports will be delivered by air parcel post, and organizations will be invoiced separately. 48-hour Datapost delivery can be arranged to most cities, at extra cost, if required.

Reprinted by permission from

Technology Transfer News

Bug-Free Software

an interview with James Martin by Leonard Kleinrock as they flew over the Bermuda Triangle

Leonard Kleinrock for TTI: We've learned to live with software which occasionally behaves in mysterious ways—in unpredictable fashions. You tell us you've identified a software technology which generates mathematically provably correct software. It seems that what you're saying adds up to an enormous revolution in the way programs are going to be generated and written.

James Martin: I think that when historians in the 21st century look back at the extraordinary history which we are creating today in building up the computer industry, they're going to be quite amazed that we would have dared to write complex operating systems or standards for local area networks using an animal-like brain, which is completely incapable of generating error-free complex logic. So it's my view that this technique, or some other rigorously based technique, is extremely important in the history of computing and that it is going to completely revolutionize everything that we do.

TTI: Yet, university professors claim that such a development won't take place in a practical way until the 21st century. Is what you have uncovered a system that actually works?

JM: Yes, absolutely! As you know, I wrote a book on the subject, and in writing the book I had to create quite a number of programs as illustrations, and they were easy to create. The system is highly usable, and it does create programs which are mathematically provably correct.

TTI: Aren't the examples you refer to discounted as toy programs?

JM: Because the ones that I wrote for the book were tutorial, they didn't have the complexity of "real-life" programs, but the technique has been used for programs which are extremely serious, for example, a radar scheduling system on a ballistic missile.

TTI: That certainly would require exact operation with no possibility of failure . . .

JM: And it's an exceedingly complex algorithm!

TTI: Why is it so difficult to prove the correctness of software as we know it today?

JM: Because in complex software there are vast numbers of possible control paths. Suppose one took a very simple example, i.e., the picture in paragraph 3.2 of my book with only 7 blocks. For this simple example there are over three trillion unique paths through the module. If it were possible to test one of them per milli-second, the total time for a complete test would be a hundred years! Even if one could do a complete test like that, we wouldn't know when we had detected an error. So, as one gets a buildup of control paths, program testing becomes almost impossible. Now this is a technique with which we definitely could not employ dynamic testing. That would be replaced by upfront static verification which would prove that the logic is correct.

TTI: This is one of the few times in my career when I've heard of a static approach proving superior to a dynamic approach, and yet it certainly seems to be the case.

JM: If you look at complex projects, more than half the projected cost is consumed by de-bugging. For example, Fred Brookes, in his famous book, "The Mythical Man Month," gives an estimating rule that one should allow at least half of the elapsed time for program and integration testing. So you can imagine if all that testing disappears, it's going to cut your schedule enormously. With this technique, not only will the testing disappear, but the programming is going to disappear as well because the technique is, in effect, a specification language. And what you're creating are provably correct specifications and from those specifications automatically generated code which would execute.

TTI: Are you really saying that once one has specified the task with this package, there's no coding to be done?

JM: No, one could better describe this as an application generator. We now have many application generators on the market, the majority of them able to generate only

specifications of relatively simple applications, but here we have an application generator which is completely generalized—where one could create specifications for exceedingly complex logic. One could create specifications for any type computing system and then automatically generate a program code which is provably correct.

TTI: Ah, . . . but isn't there a trap there? How do we know that the program which generates the code is correct?

JM: Because the program which generates the code is created with the program which generates the code! That's a bootstrapping function.

TTI: We haven't really described what this bug-free system is and just how it works. You began discussing it earlier with the specification. Maybe you can tell us in simple words just what it is and how it does its magic.

JM: One can represent any function by writing y = f(x) and draw a picture like the following:

Now, suppose your function is very complicated; suppose, for example, your function is to generate an electronic transfer system. You'd have certain inputs and certain outputs. That one block shown above is an overall statement of requirement, without any precision. We now want to make that precise by decomposing that function into sub-functions, and we wish to keep on decomposing it until we've got a statement of sufficient precision to be able to generate code. All of this is done with a mathematical rigor which enforces us to have a specification which is *mathematically provably correct*. One of the attractive features about this is that your very broad statement of requirements is made in the same language, with the same type of diagram, as your more detailed specifications. In other words, there's only one language. If one utilizes the traditional DP development process, we start off by writing requirement statements in English; we might then represent specifications with the aid of dataflow diagrams; we then convert those into hierarchical structures representing program design; and we then convert those into COBOL! So, in traditional systems, we've very often got four incompatible languages. Using this design procedure, you've got only one language, and one type of graphics representation, which takes us all the way from the very broad statement of requirements down to a totally debugged system.

TTI: But how are changes handled deep down in the detailed specification?

JM: It's wonderful to handle changes! It's just a dream for maintenance. And as you can imagine, in writing a book, you make many, many changes because you want to get tutorial examples. Each time I think up a better way of writing a paragraph, I am forced to change the program to which the paragraph relates. The examples in the book were changed many times, and as I change a block that's lower down on the chart (in other words, a more detailed block), I get an indication flashing on the screen of all consequential changes which must be made in order to preserve a system without any bugs in it. The problem with making changes in complex systems is that, as one makes changes, it sets off a vast chain reaction of other things which have to be modified. Usually, you don't spot all of those other things, and the cost of making a seemingly trivial change becomes very expensive because it creates an enormous crop of new bugs which have to be located. Here, as soon as you make any change, it automatically tells you of all consequential changes.

TTI: It seems that a lot of its strength comes from this common language. In the usual programming methodologies, if I change a requirement spec, I must then ask myself, "What are the implications down at the coding level?" Here, that's not necessary.

JM: Yes, if you look at the typical maintenance situation, you encounter a programmer under severe time pressure to make a correction to a COBOL program. What one really ought to do is change the structured chart of the program and then change the dataflow diagram and then change the English text which relates to it—and one almost never does that, due to lack of time; so, the code which exists becomes successively patched in a way which doesn't match the higher level representations of the program. With *this* system, when you make a change, the representation right from the top to the bottom is automatically being changed. Furthermore, you do it at a screen, and you can do it very fast. One of my first reactions to this whole piece of software was that it is a lovely toy to play with! But I shouldn't call it a toy because it's actually a very complicated system. It is like the best CAD/CAM systems or the best systems for doing architects' drawings on the screen. It's fascinating to use in that you have context diagrams, and when you make one change to the diagram, many other things happen. One can make changes in an elegant fashion, very fast.

TTI: Surely, you can't view the entire structure that you have generated on one single screen?

JM: No, you'd have many interconnected screens and a way to navigate through them.

TTI: So, if I wanted to identify all the places where my change had caused further changes, I'd have to scan around in this larger space?

JM: Not really. The reason is that the system won't let you generate any code until you've caught all of the errors which your changes had triggered.

TTI: But I thought one doesn't produce errors with this system?

JM: It generates bug-free code, but in order to do that, you have to get bug-free specifications. Now, you certainly can put something on the screen which is inaccurate, and it's going to indicate to you that there is an error in your human statement which causes it to be noncomputable.

TTI: What kind of errors will it catch—contradictions, inconsistencies, omissions?

JM: Yes, contradictions, and yes, inconsistencies, sometimes omissions.

TTI: How does this system handle documentation?

JM: The documentation is automatically built as you build the specifications; it's a beautiful example of a self-documenting system.

TTI: Are changes immediately reflected in an updated version of the document?

JM: Yes, and along with each block which represents the specifications, we can put some explanatory English language text.

TTI: Very good! You have a recursively correct system. Jim, are you predicting that all existing application generators will be replaced by this new bug-free system?

JM: No, because there is another requirement in what one might call a computable specification language. As well as the capability for non-procedural representation with which one can get results very fast, one also needs extreme user-friendliness. For simple systems, there is certainly going to remain a substantial market for user-friendly application generators.

TTI: Are you saying the price we pay for bug-free systems is that they are not user-friendly?

JM: No, This particular one is user-friendly, but it is more appropriate for creating systems with complex logic than for generating for example, a simple report.

TTI: How do these bug-free systems compare with the traditional structured approaches to program generation?

JM: Most structured approaches being taught in almost all the courses offered on the subject in the United States do not even attempt to create bug-free code and certainly do not succeed in creating it. If one looks at the diagrams drawn by the practitioners of that technique, it is possible to analyze those diagrams and re-do them with this technique. As that is being done, it becomes evident that typical structured diagrams are *absolutely full of errors.*

TTI: That sounds like a wonderful challenge to our readers. Can they put some of the existing structured programming designs to the test of this bug-free design procedure and see if, in fact, they do contain errors? That leads to another question of interest to our readers—how long does it take an individual to learn to use this system?

JM: I think an intelligent individual, including one who has not been involved in the computer industry before, could learn how to get valuable results with it *in two days.* But before one became an expert, like anything else in computing, many weeks of practice would be needed in actually building systems.

TTI: So, early on, one could generate some simple systems and then, as he matures, develop much more complex systems.

JM: Yes, there have been lots of examples of fairly bright people learning how to generate simple systems with this in two days.

TTI: How long did it take you to learn to use it?

JM: Far less than two days!!

TTI: Would the naive, non-DP user require a DP professonal to assist him in this process?

JM: Not necessarily. But I think it would be better to use the system in typical DP installtions and have a professional (not necessarily a programmer but certainly someone who is very competent and capable with the system) helping the end-users. I tend to regard it rather more as a tool for the DP professional than as a tool for the end-user.

TTI: Is there currently available to the user-community an implementation of such a bug-free system?

JM: Yes, in fact, there is. It's produced by a company called Higher Order Software in Cambridge, Massachusetts and the name of the product is USE. IT® . And it runs, today, on any VAX computer.

TTI: Is that the only system on which it is implemented?

JM: At the present time, yes, but it can generate code which is portable to other systems.

TTI: For which languages do we currently have USE. IT compilers?

JM: It can now generate code in FORTRAN or PASCAL; they are creating a COBOL generator and they have plans to create an ADA generator.

TTI: What is the status of the documentation on this product, USE. IT?

JM: It's pretty good documentation. One can learn how to use it from the documentation—especially if you read my book on the subject!

TTI: Is your book currently available?

JM: Yes, it's entitled PROGRAM DESIGN WHICH IS PROVABLY CORRECT, published by Savant in England.

TTI: How can we learn more about this product and this technique?

JM: Read the book, and/or get a demonstration of it from HOS. But the best way to learn more about it is to find some program of reasonable complexity and teach yourself how to build that system with this tool.

TTI: The entire development of the bug-free software you have just described is enormously exciting! Once again, it is an example of American creativity at its best. To take full advantage of the technique, it should be made widely available. And therein lies the rub! Once it's publicized, for example, in your book, do we not run the danger of handing over our latest technological developments to our foreign competitors, in particular, the Japanese! After all, they have not been shy in adopting our technology and then dominating the market place.

JM: We've been protected to some extent from Japanese hardware by the fact that the Japanese software in the past has been so bad. The question is often asked, "Are the Japanese likely to improve their software production capability in the same way that they have improved their hardware production capability?" And one of the things that we might have reason to be alarmed about is that, if the Japanese really take off with this technique, it would tend to fit their personalities very well, I think, because they get on very well with rigorous mathematical discipline techniques. This would enable them to create exceedingly complex software which is completely bug-free and easy to maintain.

TTI: That's a frightening prediction, Jim. One wonders what we can do in that case to protect ourselves.

JM: We've got to keep ahead!

TTI: Are you suggesting, therefore, that our armies of programmers should now retrain themselves in these new technologies not only in application generators, but in this particular bug-free approach?

JM: Absolutely! No question whatsoever! The ordinary programmer ought to understand that the computer world is changing, and if such programmers want to earn high salaries in the future (as well as be on the vanguard of progress), they've got a lot of new learning to do—of techniques that will make them, as individuals, more valuable, more powerful and capable of earning those high salaries.

TTI: Inevitably, that means the universities will have to adopt this new approach. Do you think they will?

JM: In twenty years' time.

TTI: Now, now, Jim, you're talking to a university professor!

BBJ
Boston Business Journal
August 20-26 1984 Volume Four, Number Twenty-Five 75 cents

Draper Lab, situated at Ground Zero of the antinuclear movement, still attracts protesters, but is looking to lessen its reliance on weapons-making. photo by Ellen Shub

CORPORATE PROFILE

Draper Lab tries softening its image

With strategic weapons, a computerized sewing machine

by John Strahinich
Journal Staff

Deep inside the labyrinthine corridors of the Charles Stark Draper Laboratory in Cambridge, engineers are fiddling and diddling with the MX missile, the Trident II missile, the Space Shuttle, various Star Wars weapons—and a sewing machine. The missile and space work make up the bread and butter of an outfit best known for designing precision guidance systems that can stick a Minuteman in the hip pocket of a Muscovite taking a stroll in Red Square. The sewing machine is a different matter.

It is a computerized machine that can fold and sew limp pieces of fabric into sleeves, vests and the backs of suit coats with a speed and precision few garment workers can sustain. For the hard-hit US apparel industry, the Draper machine represents a chance to gain a critical edge in its attempt to stave off competition from foreign manufacturers, especially those in the Far East. For Draper, the machine represents an attempt to diversify, lessening its heavy reliance on the Department of Defense, by far its biggest customer.

It is also an attempt, Draper officials admit, to tone down its image as a weapons lab in a state that has become the hotbed of the nuclear-freeze movement. Last November, for instance, Cambridge residents voted down, by 60 percent to 40 percent, a referendum that could have forced on Draper a choice between the unthinkable—canceling its military contracts—and the unspeakable—relocating its facility. Draper officials took little solace in the outcome. The referendum itself was the message, and the message was clear: Draper Lab is situated at Ground Zero of the anti-nuke movement.

"From a public relations standpoint, our primary customer is the federal government—the strategic weapons"

(see DRAPER, page

From lessons learned at Draper

A pioneer mission into 'error-free' programming

by Rosemary Hamilton
Journal Staff

As Neil Armstrong prepared to step on the moon's surface that July night in 1969, a shot of fear raced through Margaret Hamilton's body more than 250,000 miles away. Hamilton and her team of software engineers had gathered to watch the historic moment on television screens at Charles Stark Draper Laboratory in Cambridge. Of the millions of people waiting for Armstrong to take that famous step, only this group saw the error signal flash on a monitor. Something was wrong with the software on board the Apollo 11 spacecraft.

"Our faces went white," recalled Hamilton. The error signal persisted. No one could figure out what had gone wrong. Neil Armstrong was stepping out of the Eagle. The room at Draper Lab went still. Suddenly the signal stopped. At 10:56 pm, Armstrong stepped on the moon and the Apollo 11 mission was completed without a hitch.

The software, said Hamilton, saved the mission. One of the astronauts had been following an inaccurate checklist of computer commands. Too many commands were being keyed into the computer and the system was overloading. But the software had been designed to prioritize incoming data. It was able to throw out the low-priority commands and focus on the essential tasks. The error signal, though


Margaret Hamilton of HOS: automating the industry that automated industry. Ivan Akins

unclear, was the software's way of saying "No more!"

Margaret Hamilton's team was lucky that time. But it was knowing that luck eventually runs out that brought Hamilton to where she is today. The 47-year-old mathematician now heads Higher Order Software, Inc. (HOS) in Cambridge. The company recently introduced its first product, a software engineering tool called USE.IT that is designed to make error-free software.

The product, based on a mathematical theory, revolutionizes the way software systems are made. It completely eliminates the current way of designing, writing and producing software. The in-

(see HOS, page 7)

Current yield below national median

City looks to state to manage its pension fund

by Sue Reinert
Journal Staff

The city's Retirement Board has voted to take its pension fund of $325.4 million out of the hands of four private money management companies and give its business to a new state agency, in hopes of earning more on its investments. At present, the fund underperforms a national sampling of public pension systems. But there are legal and policy questions that could block the switch. In fact, one issue involves a potential loss of $500 million for either the state or municipal pension systems throughout Massachusetts.

Two weeks ago, the three-member Boston board decided quietly to join the newly created state agency, the Pension Reserve Investment Trust Fund (PRIT). The fund, to be supervised by a board of nine members, was created by the state Legislature last December in a move to modernize municipal pension sytems in the state. About $400 million in a reserve fund for state employees and teachers will automatically go to PRIT, and other municipal systems may join if they want. The main pension funds for state workers and teachers, which held assets of about $2.3 billion as of last Dec. 31, will not be part of the new fund.

The state agency still has no office and no investment managers, but it does have one big problem that has become pressing because of Boston's desire to join. The problem is: how much are the assets of municipal pension funds such as Boston's worth? Many of the investments held by Boston and other systems are old bonds paying low interest, which makes them worth much less than their face value in today's market, according to Phillip D. Kett, chief investment officer in the state treasurer's office. Their market value is lower than their book value because investors insist on a deep discount when buying low-interest bonds in order to increase their effective yield.

Kett said that statewide, the gap between the market value and the par value of pension fund assets has been estimated at $500 million.

The members of the Pension Reserve Investment Management (PRIM) board are now trying to decide which figures to use—book or market value—when they accept pension fund assets from Boston

(see PENSION, page 5)

INSIDE:

0 74470 16693 34

Curtain falls on Puritan

by Geoffrey Gevalt
Journal Correspondent

Puritan Furniture is no more. Last week, US Bankruptcy Judge Harold Lavien approved sale of all its furniture to Barber Sales Co., a New Hampshire auction company, for $800,000 plus 15 percent of the proceeds when Barber auctions the furniture.

The sale came after four heated court hearings in which three other companies tried to purchase Puritan's furniture, estimated to have cost upwards of $2 million wholesale.

The court action comes despite some question as to who exactly owns the furniture. According to court documents, much of it had been purchased by Furniture Distributors Inc. (FDI), a Tennessee liquidator that had been brought in by the Norwood furniture company to run a close-out sale to raise much-needed cash. According to court testimony, no one has any accurate accounting of who owned exactly what. The court-ordered sale, however, gives Barber clear title to all the furniture.

The entire legal matter began a month ago, when several furniture distributors attempting to reclaim furniture they had sold to FDI put the Tennessee company into involuntary Chapter 7 bankruptcy. FDI turned around and put Puritan into Chapter 7.

Stanley Adelstein, owner and president of Puritan Furniture, said he could not comment on the sale because he had not had a chance to review it. Efforts to reach his attorney, Gerald Rosen, were unsuccessful.

Judge Lavien has not yet ruled on how the proceeds of the sale will be distributed—an issue that is likely to be hotly contested in future court hearings. According to court documents, Puritan owes some $5 million to furniture companies, leasing companies, banks and distributors.

According to several lawyers involved in the case, Barber will be conducting a public auction of the furniture.□

HOS

(continued from page 1)

dustry that has been busy automating other industries is on the verge of being automated itself.

The mathematical theory is based on a study done by Hamilton and her colleague Saydean Zeldin. When the Apollo project began winding down in the early 1970s, they examined all the errors, or bugs, in the project's software. The two women were looking for a way to improve software engineering. "It would have been irresponsible not to go back to the beginning and see what we could have done better," said Hamilton.

And they were looking for excitement. They weren't just looking for a more logical way to make software: they wanted to be part of another major breakthrough. After all, Margaret Hamilton had helped send a man to the moon by the time she was 32. "Apollo changed my life," she said. "It had a profound affect on us. Some people never got over it. And there have been other spinoffs from Draper because of it." The follow-up for Hamilton, who was in charge of more than 100 software engineers at Draper, was going to have to be something big. She seems to have found it by starting her own business. To Hamilton, "A growing high-tech company is like a mission."

Testing the theory

With the theory in hand, Hamilton and Zeldin founded HOS in 1976. Hamilton is now CEO and Zeldin is chief financial officer. As Hamilton tells it, the theory was actually a surprise to them. "We weren't expecting it," said Hamilton, "and we didn't set out to find it." But the theory became their foundation for creating software. Since the theory was probably correct, all software designed within its logical confines of this theory would also be correct, or bug-free, they reasoned. For five years, HOS operated as a research and development firm, as Hamilton and Zeldin put the theory into practice. In 1981, USE.IT was introduced.

Its first test run outside of HOS was with the Army. The test was a success, and talk of USE.IT's potential began spreading through the industry. The word sparked investors' interest, and as HOS took USE.IT to the commercial market in the summer of 1983, the company had the backing of some heavyweight high-tech venture capitalists. In two years, the company has raised more than $9.2 million. One of the first investors was James Martin, a high-tech management guru, author of more than 30 books on data processing, and now a member of the HOS board. Other early investors were Venrock, Greylock, Merrill Lynch, Frontenac and Alex Brown & Sons. Of that $9.2 million, $5.7 million was raised last April. That round of investors included the Hillman Fund, Henry & Co., Emerging Growth, Sears Pension Trust, Cazenove, New Castle Co. Ltd., J.F. Shea Co. and Samuel Montagu Co.

Checking for bugs

USE.IT runs on Digital Equipment Corp.'s VAX minicomputers and sells for $92,000. One component of USE.IT is its language, AXES, which was developed at HOS. It is not a traditional programming language. It includes a graphics capability and "resembles natural language," said Hamilton, because a user can write functions in English phrases. The other two main components are the Analyzer and the Resource Allocation Tool (RAT). These two components are what automates a traditional programmer's work. The Analyzer automatically checks written programming for bugs when the user presses the Analyzer command button. Once the Analyzer has approved the program, another command key is hit to send it to the RAT. This component will automatically generate program code in Fortran, Pascal, Cobol or other languages. Because these two procedures are automated, a person with no programming backround could write software on USE.IT.

Hamilton said USE.IT is appropriate for banking, insurance, manufacturing, aerospace and robotics applications. In addition, the software tool will eventually be designed to run on mainframe computers, the market in which HOS hopes to make USE.IT a standard. "We're going after the commercial mainframe environment, the disciplined users, people developing complex systems," she said. "They understand that the old lifecycle [that is, the current software development process] is wasteful."

Typically, the software development process begins with a list of requirements for the software drawn up by the potential user. In a corporation, for example, the head of the mailroom would send a list of requirements to automate his department to the data processing manager. From there, the specifications for the electronic computations that the computer will perform to accommodate those tasks are drawn up. The design, programming and integration steps follow. All of the programming is integrated as a software system. Then it is tested.

The final stage of this process is maintenance—getting the bugs out. This generally eats up most of a software project's budget because the steps that went into creating the program are not necessarily error-free. The beginning stages are not governed by strict rules of logic. Programming is often done in an ad hoc manner. For instance, mistakes can occur in the initial requirement stage. The instructions from the mailroom to the data processing department may be unclear. The result is a system that wasn't designed to perform the functions that it was supposed to perform. In addition, manual processes are used in between phases, meaning that even illegible penmanship can be responsible for errors.

Defense department tests

A critical flaw in the traditional software process is the testing. Since it is near the end of the cycle, errors made early on are hard to find. They often become the building blocks for other errors. James Martin, a board member at HOS, aptly refers to today's complex systems as minefields. In his book "Program Design Which Is Provably Correct," Martin uses an example of a fairly simple software program with 12 iterations. There could be more than three trillion paths, or possible functions, that the software could follow, he said. "If it were possible to test each of them in a millisecond the total time for a complete test would be over 100 years," Martin concluded.

As an alternative to the impossible, a method of dynamic testing, or testing as many possible paths as time permits, has become standard. Yet there is no guarantee that what is tested and found to be bug-free is representative of what lurks untested in the program. This built-in risk increases the cost of maintenance in proportion to the complexity of the project.

When USE.IT was introduced, HOS performed comparison tests of software work done with and without USE.IT. The tests were based on work done by programmers at the US Department of Defense. According to Hamilton, these programmers are required to produce 10 lines of program code per day. Three days of programming work, HOS concluded, would take only four hours with USE.IT. The difference was based on the average amount of time spent on maintenance per line of code compared to USE.IT's automatic testing. Humans may not be able to test the millions of possible paths, or functions in a software program, but computers can. In the defense department example, Hamilton said USE.IT increased productivity by 600 percent and cut costs by 83 percent.

The key to USE.IT is that it traps bug-ridden software before it gets to the user stage. The Analyzer won't approve a faulty program. And the RAT won't accept it to be translated into program code unless it is logically correct. As in the

(see HOS, page 8)

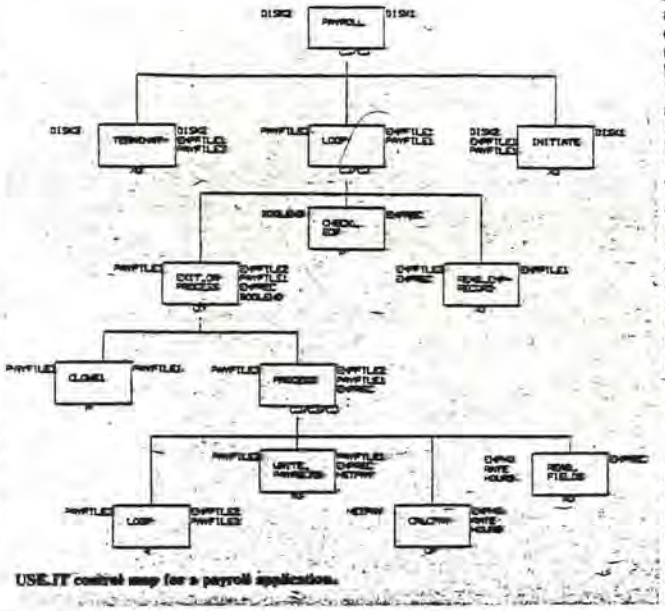
HOS

(continued from page 7)

game of Monopoly, a program cannot pass "GO" unless it has made a good roll. And a good roll is error-free.

Error-free program

To get this error-free program, USE.IT eliminates the step-by-step method of program writing. In simplest terms, there is no more "If this happens, then do that" type of commands. Instead, USE.IT relies on a "control map" format using the AXES language and graphics. In its most basic form, the AXES graphics display is three blocks. Each block is known as a node and represents one function of the system. The top block is the overview statement, or main function. Two blocks extend from either side of it, forming an overall triangular shape. The control map must adhere to a logical relationship. The three main relationships are "join," "or" and "include." Since each sub-block also represents a function, its task must be logically related to the main function. Additional relationships are derived from these three when working on a more complex system.



USE.IT control map for a payroll application.

The "join" relationship means the two sub-blocks are dependent on each other and the functions will be carried out sequentially in relation to the main function. The "or" relationship is a choice between one of the two sub-blocks. The "include" relationship means the blocks are independent of each other. A main function, for example, could be "Make Dinner." The subfunctions could be "Put TV Dinner In Oven" and "Remove TV Dinner From in Oven." This represents a "join" relationship, since both functions must be completed in order for the main function to be completed. The same main function, however, could be an "or" relationship. In this case, the subfunctions might be "Heat TV Dinner" or "Prepare Corned Beef and Cabbage." Only one of these two functions must be completed for the main function to be completed.

Dinner to Analyzer

"Make Dinner" can now be sent to the Analyzer. It will be returned to the user as an approved control map or as an unacceptable one. If unacceptable, question marks will appear in the location where the error was made. Since this procedure can be done early in the development of software, errors won't be built upon errors. Testing, however, can also be performed. In this case, even if a mistake was made at the beginning, the Analyzer will send it back with questions at those locations. But more important, no program code can be generated if the written program contains errors. In the traditional procedure, code would be generated, and a program would be run containing errors.

When a user is writing a complex system, that simple control map can be extended to include hundreds of nodes, or functions. In this case, it is broken down into subroutines. Simple programs can be stored in the USE.IT libraries, or memory. They could then be incorporated into other more complex programs. The simple programs have been approved by the Analyzer, so they become error-free building blocks for the more complex software.

Human garbage

Still, USE.IT can not offer a full guarantee that there will never be any errors. Although most of the system is automated, people are still sitting at terminals and entering data. The human element, therefore, leaves room for wrong information to be put into the system. But as long as the wrong information conforms to the systems logic, USE.IT will accept it. "You know," said Hamilton, "garbarge in, garbage out. But at least it will be logical garbage."

USE.IT, said Hamilton, is a "revolutionary product" that will "be introduced in an evolutionary way." If HOS has its way and makes USE.IT the mainframe industry standard, traditional programmers will be the next dinosaurs. But Hamilton said HOS is not trying to put programmers on the unemployment line. "We're taking out the drudgery of their job, and keeping in the challenge of designing software. We're actually elevating their jobs."

Yet she concedes USE.IT will probably come up against some resistance. Today's software engineers will have to rethink the way they perform their jobs. And these engineers who have made a career of automating other people's jobs may find it difficult to do the same to themselves.

But Hamilton said that corporate managers who have felt the pinch of high maintenance costs will readily accept USE.IT. The client list currently includes Raytheon, Digital Equipment Corp., Data General, GTE, Honeywell, Wang, Texas Instruments, Computervision, DuPont, Eaton Corp. and General Dynamics. In the future, Hamilton said HOS will enhance USE.IT to run on personal computers linked to mainframes. "Right now," she said, "we're making an effort to put ourselves out there. But our goals are ambitious. We want to be to software what IBM is to hardware."□

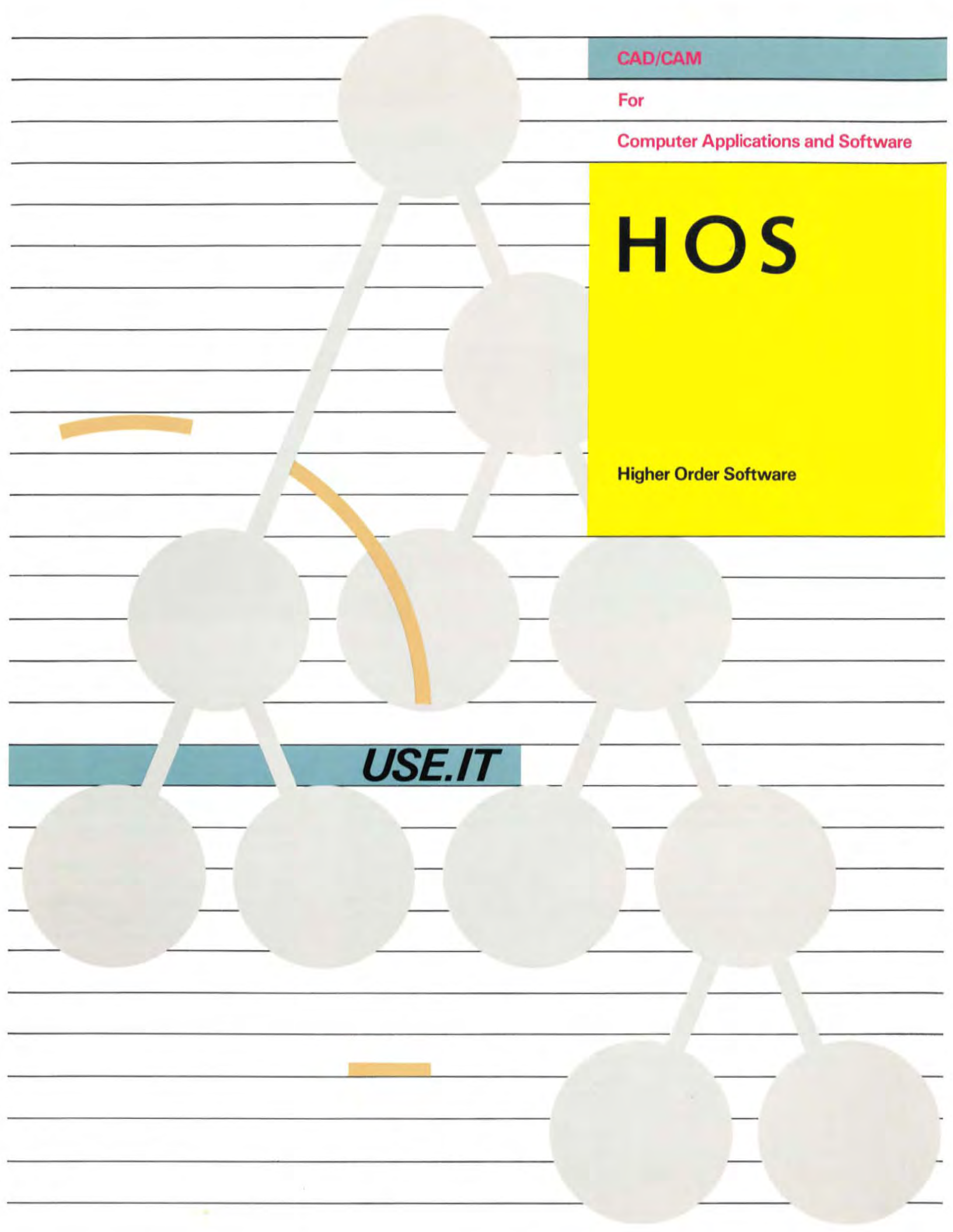
CAD/CAM

For

Computer Applications and Software

HOS

Higher Order Software



USE.IT

Software In The Modern Corporation

In today's corporation, the tasks facing the data processing department are more complex than ever before. It is becoming clear that the ability of a company to compete depends largely on its ability to process information more efficiently than its competitors. This means that the DP staff must not only maintain a myriad of existing programs, but also cope with a large and growing backlog of new applications, just to maintain the competitive edge of the corporation.

These new applications, which can range from an insurance claims system, to an order processing system, to a financial control system, are based on company strategic requirements whose complexity is often intimidating. Transforming these requirements into reliable software is time-consuming and expensive at best, and in some cases impossible, using current development tools.

For the successful corporation, resolution of this problem is not a luxury; it is a necessity. To compete, companies must be able to produce reliable software, and produce it quickly.

There is a demand for a logical method of producing this software, and for a tool that enforces that method.

Such a method, and such a tool—USE.IT—are now being introduced to the IBM marketplace by Higher Order Software, Inc. USE.IT is designed for IBM software and applications developers who need to deliver systems that are demonstrably reliable, regardless of complexity; and who need to deliver them more quickly than ever before.

USE.IT—CAD/CAM For Computer Applications and Software

USE.IT is a technological breakthrough. A computer-aided software design and production tool, it is analogous to those computer-aided design and manufacturing (CAD/CAM) tools successfully used in industries ranging from aircraft manufacturing to micro-chip design. Just as CAD tools let the designer model a product and correct design errors before production begins, USE.IT lets software designers create specifications that are logically consistent and complete before a single line of code is produced. And just as CAM tools automate production, USE.IT automatically generates running programs.

The need for such a tool is evident. Traditional development methods are no longer able to satisfy the demands of the modern corporation: information systems are developed according to schedules bearing little relation to production realities; application programs are obsolete before they are shipped; vast groups of employees are assigned to tinker with these presumably finished applications for decades; finally the accumulated weight of their fixes causes systems to collapse beyond repair.

The statistics are alarming:

- From 50 to 70 percent of data processing budgets are allocated to application maintenance.
- Six generations of programmers maintain a program throughout its lifetime.
- It is 10 to 100 times more expensive to correct an error in the testing or maintenance stages than in the requirements or design stages.

These problems are the direct result of the lack of a usable tool that *enforces* a formal and rigorous development method. This lack ensures the continuation of enormous application backlogs. These backlogs threaten to paralyze even the maintenance of existing systems, let alone the creation of new systems vital to the company.

Current Productivity Tools

Currently available productivity tools, such as database management systems, fourth-generation languages, report generators, code generators, and screen painters, automate the programming phase of software development. It is a fact, however, that the source of the great majority of errors (and of the resulting unreliability of software systems) is not the programming phase, but the requirements and design phases. These tools simply do not address requirements and design, which are the primary problems.

A tool is required which is usable throughout the life of an application system. Such a tool must support:

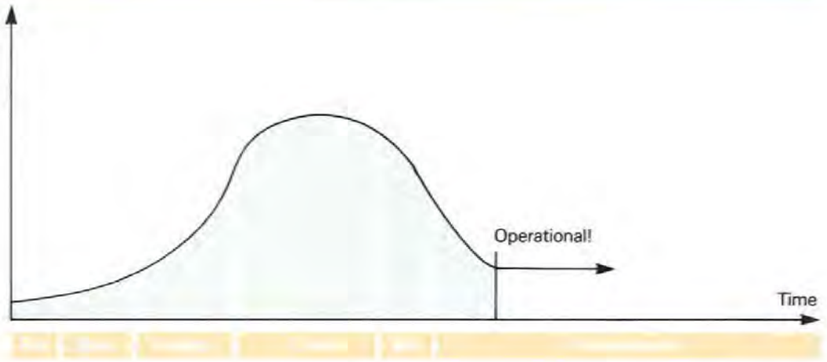
- Detection of errors in the early stages of the development process, where they are easiest (and cheapest) to fix.
- Integration of the development process by a rigorous and traceable linkage between requirements, design, specification, and implementation.
- Insulation of business knowledge from computer knowledge, by keeping descriptions of *what* the system does in business terms separate from descriptions of *how* it is done on the computer.
- Automated prototyping capability, so that the user of the system can actually run the system, and know that it meets his requirements *before any final code is written*.
- Automated generation of final production code, so that code is guaranteed to match the specification.
- A formal link between the running system and new enhancements required by a changing business environment. With such a link, systems will no longer become degraded in the maintenance stage, and can endure indefinitely.
- Automated generation of system documentation, so that documentation is always current and correct.
- Automated management of reusable code and data. Re-use of existing programs and structures can significantly reduce both delivery time and cost of new systems.

Higher Order Software, Inc. has developed a methodology that meets these requirements, and has implemented it in USE.IT for the IBM software market.

The value of USE.IT has already been proven on the Digital VAX architecture. There it has been employed to design large, complex software systems where reliability is critical. Now the developer of IBM software systems can benefit from this proven product.

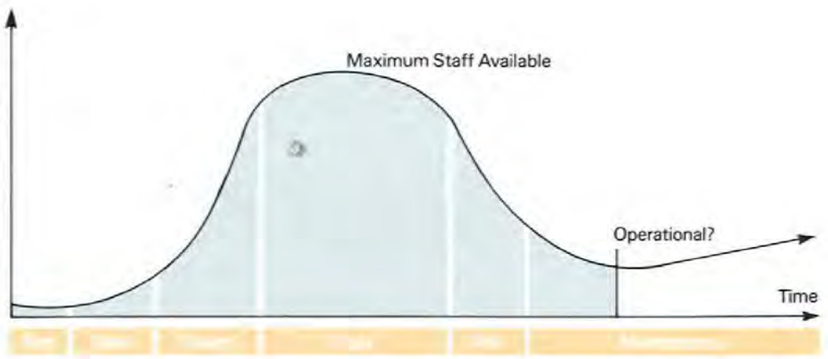
Typical System Development Plan

During project planning stage, standard life cycle techniques are used to estimate time and resource requirements.



Actual System Development Experience

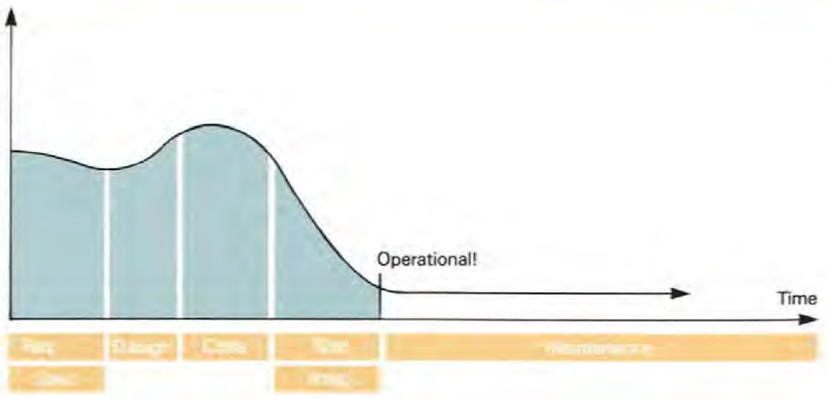
As new and unclear requirements emerge, the best of plans give way to slippage and added resources. Functions and features are deferred to the maintenance phase.



Actual USE.IT Experience

The discipline and power of USE.IT yields

- *Reduced resource requirements*
- *Completion far ahead of standard life cycle schedule*
- *High quality results which all but eliminates maintenance effort*



The HOS methodology provides an explicit set of rules and procedures with which to design and implement complex software systems.

Systems are designed at a terminal using a graphics editor; the designer describes the system as a set of functions arranged in a tree structure. Functions are depicted as nodes on this tree and are progressively decomposed from the highest level general business functions down to primitive operations that can be implemented on a computer. The data each function uses and the data it produces are also shown in the tree diagram.

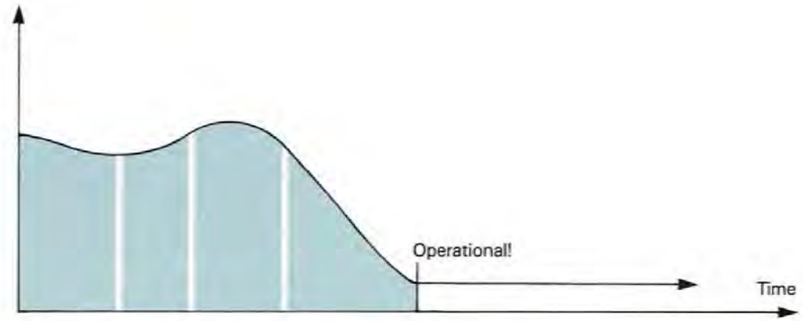
Detection of Errors

At any stage in the development process, the system specification can be automatically analyzed for errors, and necessary corrections made immediately, when changes are least expensive.

The Impact of USE.IT As a Tool For Managing Reusable Code and Data

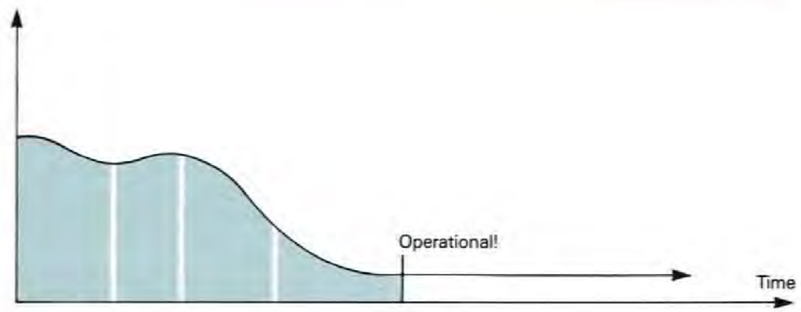
Additional *First Use* Benefits

- *Establish catalog of key code modules and data elements*
- *Create management standards for reusability*



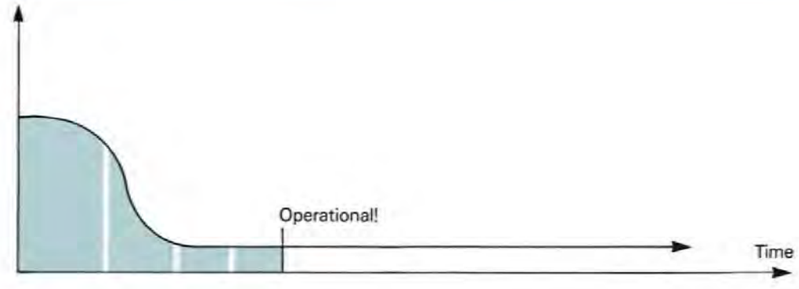
***Second Use* Benefits**

- *Incorporate previously tested modules and data*
- *Increase library with new modules*
- *Reduced resources and time to create successful system*



***Nth Use* Benefits**

- *Major time and resource saving due to reusability of ever growing library and data base.*
- *Continued contribution of new modules and elements.*



Integration of Development

Every stage of the development process, from requirements to the final design, is expressed in the same tree diagram. There is, therefore, a direct and traceable passage from requirements down to the final program code. This eliminates the problem of translating systems descriptions from one stage to another, and allows the designer to trace the functionality of his system from requirements straight through to the final production code.

Retention of Business Knowledge

When creating a system description with USE.IT, one begins at the higher levels of the tree by describing the actual business functions to be performed by the system; it is only when one gets down into the lower levels of the tree that the computer functions that actually do the work are described. That essential business knowledge, which is the heart of the corporation, is therefore clearly described and saved, and is always available. If it becomes necessary to move this application to another machine or another environment, that essential knowledge does not need to be restated. Thus, the enormous dollar investment in software products and custom applications is protected.

Automated Prototyping

Some productivity tools require that prototypes be built manually. Systems created with USE.IT are working systems from which working prototypes can be *automatically* generated at any time in the development process. Specifications can be "run" at any time in order to determine if they in fact do what the user wishes them to do. This capability, together with automated analysis, provides the developer an unequalled ability to verify that the system, *as it is being developed, and before any code is written* is doing what it is supposed to do, and is doing it correctly.

Thus, management can count on the existence of an application solution that always matches the current business requirement. Reliable test results of the software being developed can be delivered at all stages of the life-cycle—not only at the end, when code has been produced.

Automated Code Generation

USE.IT will automatically generate production code for any USE.IT specification that has been analyzed and found to have no errors.

Formal Link Between Running Systems and Enhancements

If, in the maintenance stage, changes have to be made to the system, they are made to the specification, and new code is generated to match the specification as modified. Thus, the specification and code always match, and the system does not degrade over time as revisions are made.

Automated Documentation

USE.IT will also generate English-language documentation for any USE.IT system. This unique capability ensures that any program is correctly documented at all stages of development. This makes it easier for multiple teams to work on a project and simplifies maintenance of existing programs.

Automated Management of Reusable Code and Data

Re-use of existing, reliable code and data is possible only if they are correctly interfaced with the new system being developed. USE.IT guarantees the correctness of these interfaces, and manages the integration of these reusable modules into new applications.

USE.IT Version 1

USE.IT Version 1 runs under the MVS operating system and supports an interface to both IMS and CICS. Device support includes the 3278/79 or compatible terminals. Output can be directed to a range of plotters and printers.

Future developments of USE.IT will further implement the HOS methodology, and will continue to add the functions required to meet the changing needs of users.

USE.IT and The Future

USE.IT is a technological breakthrough that enforces a rigorous and formal software development method. It provides automation of error analysis, prototype generation, code generation, and documentation. This combination of formal rigor and extensive automation provides a realistic solution to the challenges facing the information profession.

USE.IT can break the chronic backlog of applications that current tools cannot begin to address. It enforces consistency in both initial system design and subsequent enhancements, thus solving the expensive maintenance problem. It frees the developer from chasing interface bugs, syntax errors, and dataflow errors, and allows concentration on conceptual correctness and dynamic testing.

But USE.IT can be even more than an operational tool for the DP department. With its rapid prototyping capability, it can become a strategic business tool. Management can quickly determine if proposed applications will work as envisioned; modifications can be made easily as situations dictate. The entire development cycle becomes more flexible and responsive to management needs, and the corporation becomes more flexible and responsive to the marketplace.

Better software, faster. USE.IT.

Headquarters

2067 Massachusetts Avenue
Post Office Box 531
Cambridge, Massachusetts 02140
617-661-8900
Telex 951253 HOS INC CAM

Eastern Region

551 Fifth Avenue
Suite 1110
New York, New York 10176-0027
212-490-8721

Western Region

8445 Freeport Parkway
Suite 420
Interfirst Place
Irving, Texas 75063
214-257-3758