XCP: An Experimental Tool for Managing Cooperative Activity

Suzanne Sluizer and Paul M. Cashman
Intelligent Systems Technologies Group
Digital Equipment Corporation
77 Reed Road, HLO2-3/C07
Hudson, MA 01749

Abstract

A project usually requires the cooperative efforts of more than one person to accomplish its goals. As the number of people working on a project increases, the time spent in coordinating their efforts multiplies, and difficulties often arise. Communication breakdowns cause major problems because communication is the cornerstone of effective cooperation. Decision making becomes complicated because areas of responsibility are ambiguous. Procedures which are set up to ensure critical actions occur often degrade over time. *XCP* is an experimental coordinator tool which assists an organization in implementing and maintaining its procedures. Its goal is to reduce the costs of communicating, coordinating, and deciding by carrying out formal plans of cooperative activity in partnership with its users. It tracks, prods, and manages the relational complexity as captured in the formal plan, so that human resources are available for more productive tasks. It can aid in the training of new staff because they do not have to learn a procedure in an ad hoc fashion. A prototype has been implemented in the VAX LISP™ language, and runs on a VAX™ 11/785, 11/780, or 11/750 processor under the VMS™ operating system.

1. Introduction

Consider a typical project. It has more than one team member, so tasks must be subdivided and assigned to individual team members. Because of the dynamic and interrelated nature of these subtasks, it is often difficult to keep track of what each team member is supposed to be doing, and some subtasks may be forgotten. Another factor which compounds these difficulties is that an individual team member may be assigned to work on more than one subtask. Such an individual must keep straight the functions and responsibilities of each subtask, and can be easily overwhelmed by complexity as the relationship among subtasks becomes more intricate.

Communication is essential to the success of multi-person projects. A change in one individual's work assignment or schedule may significantly affect other team members. Projects institute weekly meetings and regular progress reports to ensure that critical communication occurs. However, these techniques often must be supplemented by spontaneous meetings and informal conversations at the coffee machine. [Peters83] argues that the "transaction costs" of communicating, coordinating, and deciding have been vastly underestimated. These costs arise from a geometric increase in complexity associated with arithmetic growth in numbers of employees, if they need to interact to get tasks done. Simply put, transaction costs rise explosively with the number of people who must work together on a task.

An example of this coordination problem as it arises in software maintenance is as follows. One problem report may in turn be subdivided into a sequence of problems to be parceled out to various workers. A given problem may have to be handled as a sequence of fixes in successive releases. Conversely, a given fix may take care of several different problems.

In addition to these complex static relations among objects, there are dynamic processes overlaying the task structure. The problem must be classified, its priority set relative to other problems, and it must be assigned to a team member. When a fix is proposed it must be assigned to a release, announced in release notes, and the user who originally reported it must be informed. The list of such possible tasks for team members is obviously quite large. At another level, the manager wants to monitor the team's activity.

Most tools deal exclusively with the configuration management aspects of this coordination problem in software maintainance by managing files [Feldman79] [Lampson83]. These approaches keep track of the relationships between the parts (i.e., files) of a software system, and use consistent parts to rebuild the software whenever changes have been made. Such tools have automated a complex process and fill an important need in maintaining software, but do not address the higher-level issue of how to coordinate the actual tasks of software maintenance. Furthermore, they are special-purpose tools and do not provide general support for cooperative activity. There are a few ongoing efforts to build environments that assist software projects to accomplish their communication and management tasks [Kedzierski82] [Holt83].

Our goal is to reduce the transaction costs of communicating, coordinating, and deciding. We accomplish this by formalizing and automating protocols, which are plans of cooperative activity to facilitate organizational procedures. Protocol execution requires the active participation of team members. For each application, a set of roles is identified that loosely corresponds to the tasks to be done, and which team members may assume. A set of objects, which represents the types of communication that may occur among roles, is also identified. Finally, we develop an automated method to support people as they assume and carry out their organizationally defined roles and responsibilities.

2. The XCP Tool

The experimental coordinator tool we will describe, *XCP*, supports and manages cooperative activity by interpreting protocols which implement and enforce organizational procedures. We believe it reduces the coordination complexity as perceived by a participant because it moves the procedure from the participant's head to a tool. Its purpose is to ensure that necessary communication takes place, as well as improve the coordination of the project members as they carry out their tasks. It also supports the assignment of multiple subtasks to a single project member.

There are several key concepts which *XCP* implements. A *person* is simply a human being. A *role* is a codification of some function or subtask. Most people assume several roles during a work day. For example, the authors between them assume such diverse roles as project leader, software engineer, lecturer, and writer. Each role is responsible for some part of the total activity, and carries with it a set of rights, responsibilities, and expected behaviors with respect to other roles. In general, the relation between persons and roles is many-to-many. An *actor* is a person who has assumed some role. An *object* is the symbolic representation of some thing; for example, in the software maintenance scenario described in the Introduction, there might be objects such as problem reports, fix reports, releases, and software modules.

XCP allows its users to define plans of cooperative activity, called *protocols*. A protocol describes the activities or tasks which make up an organizational procedure. *XCP* coordinates the actions of the project members, and assists them in carrying out that plan. It tracks activities through the defined protocol, keeps a history of all activities on a per-object basis, and informs

users of the status of each activity. A user may suspend the protocol (i.e., an object's processing) at any point and resume it later at that same point. By querying *XCP*, each team member may answer questions such as:

- On what objects am I working now (in a specific role or all my roles)?
- What is Joe's long-term work load (in all his roles)?
- What roles does Jane play?
- What objects for role R have not yet been assigned to any actor?
- Who has worked on object O? What has been done, and what must be done next?

XCP has a very simple command set. ASSUME-ROLE allows a person to assume a protocol-defined role. CREATE makes a protocol-defined object. SHOW gives information about an object, a collection of objects, a role, or a person. STATUS gives information about where in the working-out of the protocol an object is. WORK-ON resumes the protocol for some object from the point at which it was suspended. ESCAPE-TO-EXEC allows a user to temporarily leave the tool environment to interact with the operating system. LOGOUT logs a person out of *XCP*.

The workhorse command of *XCP* is WORK-ON. It activates the protocol on some object for an actor; that is, it presents the actor with a series of actions to perform, where the series is determined by the role of the actor and the state of the object. Thus, within the context of a protocol, a user has only one command to carry out the cooperative plan. In concept, WORK-ON replaces an abundance of special-purpose commands to manipulate objects. The functionality of these commands has been captured in the protocol *primitives* described in Table 1 below.

Primitive	Action
accept-delivery	accept delivery of an object dispatched from an actor in a different role
attach	attach one object to another by reference
classify	set the attribute values of an object
dispatch	send an object to an actor in a different role
wait	wait for an object to arrive in a role
you-decide	ask the actor currently working on an object a question

Table 1: XCP Protocol Primitives

Not all commands or command options are available to all users. Persons can only log in, log out, and assume some set of roles; they can perform no other actions. A role is defined by its rights and a set of actions which can be performed in a given order on a class of objects. Any person who can assume a role automatically has permission to execute its set of actions *while acting in that role*.

XCP is founded on earlier work in the area of coordinator tools, in particular MONSTR, the Monitor for Software Trouble Reporting used by the National Software Works project [Cashman80], [Holt81].

3. The XCP Architecture

The functionality of *XCP* is partitioned into terminal-specific and terminal-independent parts. It has been divided into these parts for three reasons. First, multiple terminal types could be supported with a terminal-specific piece for each terminal type, while the terminal-independent part would remain unchanged. Second, such a partitioning could allow the terminal-specific part to reside on a host different from the host on which the terminal-independent part is executing. Third, the terminal-independent part could receive its input from sources other than a terminal (e.g., a file of stored commands or another program).

The terminal-independent part has been further partitioned into application-independent and application-specific parts. The application-independent part consists of the protocol primitives and type definitions which are used to build up protocols. The primitives indirectly interact with the user at the terminal to effect certain basic actions, as discussed in the previous section. A user can respond to a primitive by "suspending" it; this means that the primitive must wait until the user "resumes work" on it to complete its execution. The application-specific part is a protocol written in terms of the primitives and type definitions provided by the application-independent part; it may also contain specialized type definitions relevant to the application. It is composed of particular roles and types, as well as functions which represent the actions of each actor.

The terminal-specific part of *XCP* is referred to as the Command Interface (CI). It is logically and physically a separate program from the terminal-independent part, which is called the Protocol Task (PT). The division of *XCP* functionality is shown in Figure 1 below; the arrow represents information flow.

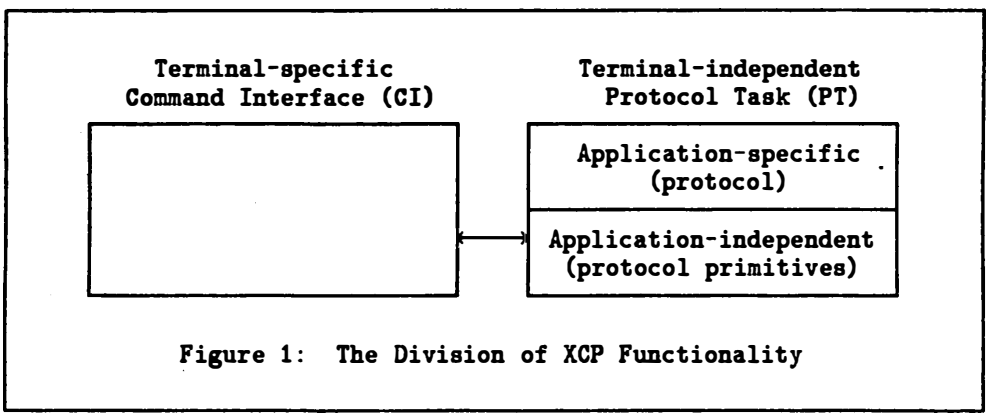

Figure 1: The Division of XCP Functionality

The CI interacts directly with a person at a terminal, and can be thought of as the user's front end. It formats the screen, performs command recognition, prints messages on the screen, and interacts with the PT through the exchange of messages. These messages represent either user-instigated commands or choices made in response to "demands" from the PT as it executes the protocol. There is one CI per active user.

The PT, when started, waits for a variety of messages. These messages represent the working-out of the protocol, and fall into five classes: commands from a user at a terminal, a user's answers to questions generated by the protocol primitives, information which must be displayed to a user, transmission of a user's answer from the protocol primitive to the invoking protocol role code, and the movement of objects from one role to another. Some of these messages are exchanged between the PT and CI; others are generated internally by the PT.

Messages are sent between the PT and CIs via operating-system-supported buffers. The PT has one buffer through which it receives messages from all CIs. Each CI has its own buffer through which it receives messages from the PT. To communicate, a process simply writes into the appropriate buffer.

Figure 2 below illustrates the architecture of *XCP*, which reflects the division of functionality pictured in Figure 1 above. Note that the terminal-specific part is represented by multiple CIs which communicate with a single terminal-independent PT. The arrows represent information flow between the CIs and PT via their buffers.

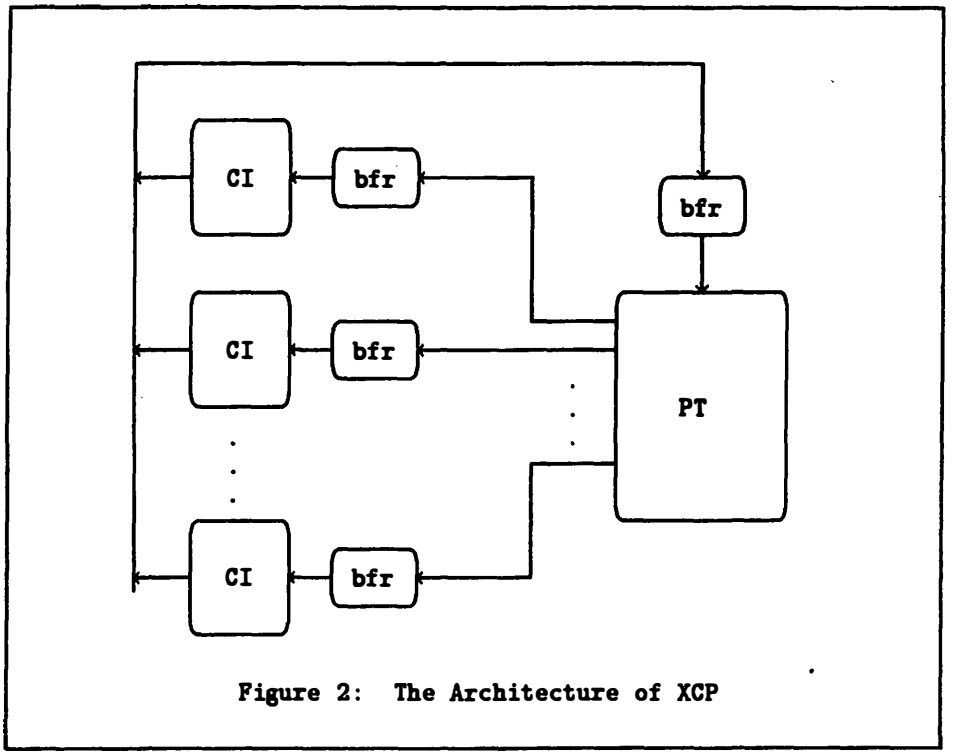

Figure 2: The Architecture of XCP

XCP runs on a VAX™ 11/785, 11/780, or 11/750 processor under the VMS™ operating system, and supports multiple concurrent users. It uses the VMS mailbox mechanism as its buffers. It is written in the VAX LISP™ language, which is based upon the Common Lisp dialect [Steele83], with some extensions. *XCP* represents persons, roles, actors, and objects with a small, frame-oriented knowledge representation language in which type hierarchies can be defined. A type is a set of named slots that can hold arbitrary values, and may inherit slots from one or more parent types. The control mechanism for *XCP* is provided by a set of functions which supports concurrent programming through message passing, and is used to build a set of protocol primitives which can be used to construct any protocol. The application protocol itself is written in a highly stylized form of Lisp, and consists largely of calls to the protocol primitives.

4. An Annotated Example

An annotated example of *XCP* usage, based on a simple protocol to manage problem reports, appears below. The protocol has three roles: USER, WORK-ASSIGNER, and DEVELOPER. Three persons, BARRETT, WINTER, and GORDON, may assume these roles as follows: BARRETT has been given permission to assume the USER role, WINTER has been given permission to assume

the WORK-ASSIGNER and DEVELOPER roles, and GORDON has been given permission to assume the DEVELOPER role. The relationship among the persons, roles, and actors is shown in Table 2 below. An actor is denoted by P@R (read person P in role R).

Person	Role	Actor
BARRETT	USER	BARRETT@USER
WINTER	WORK-ASSIGNER	WINTER@WORK-ASSIGNER
	DEVELOPER	WINTER@DEVELOPER
GORDON	DEVELOPER	GORDON@DEVELOPER

Table 2: Relationship among Persons, Roles and Actors

A USER creates a problem report (P-R in Figure 3), which is then sent to the WORK-ASSIGNER role, but cannot be sent to a specific person in that role. (This restriction is part of the USER role definition, and is enforced by *XCP*.) The WORK-ASSIGNER sends an acknowledgement (ACK) to the originating USER. The WORK-ASSIGNER then assigns the problem report either to a specific DEVELOPER (based upon such criteria as work load and problem area) or just sends it to the DEVELOPER role where any person in the role is allowed to take charge of it. The DEVELOPER proposes a solution, which is sent back to the WORK-ASSIGNER as a fix report (FIX). The WORK-ASSIGNER sends a user reply (U-R) to inform the originating USER that the problem has been handled. A picture of this protocol appears in Figure 3 below. The arrows represent objects traveling between roles, and are actually messages as described in the section on the Architecture of *XCP*.

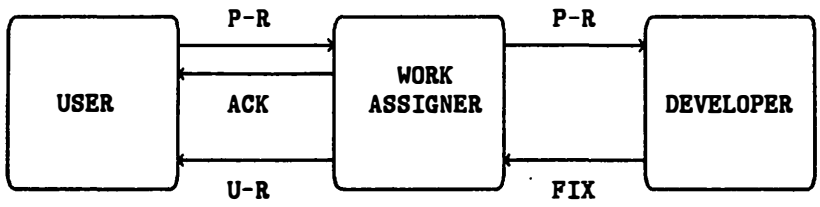

Figure 3: A Simple XCP Protocol

We will show the session as it might appear on a hard-copy device, although currently *XCP* exclusively uses the VT100™ video display terminal. We have slightly modified the wording of some output because it could not be adequately represented in hard-copy form. The user interface for the VT100 terminal formats the screen by using reverse video for emphasis, command recognition, command and options menus, and a header with information useful to the actor. A user can type "?" and receive help from *XCP* at any time.

In the example, the VMS system prompt character is $, the *XCP* prompt character is ">", *XCP* output appears in **bold**, and comments on the session appear in *italics*. Control-Z (denoted by ^Z in this transcript) is used by the user to signal "no more input" as well as "suspend".

$ XCP — *Start XCP from VMS operating system.*
>login Winter — *Identify Winter as a person.*

You can assume these roles: — *Persons can only log in, assume roles, and log out.*

WORK-ASSIGNER
DEVELOPER — *Presumably another role-player has given Winter permission to assume these roles.*

Your choice? WORK-ASSIGNER

>show person Winter

You are responsible for the following objects:

Identifier	Title	Role
P15	Program loops when given null input	WORK-ASSIGNER
P22	Unable to start program after 5:00pm	DEVELOPER
ACK14	Acknowledgement for P15	WORK-ASSIGNER
FIX16	Fix report for problem report P5	WORK-ASSIGNER
UR25	User reply for problem report P7	WORK-ASSIGNER

There are four types of objects of interest here. Problem reports describe an anomaly in the behavior of some program. Acknowledgements assure the user reporting the problem that the developers will work on it. Fix reports describe a specific solution to the problem reported. User replies report the solution to the user who originally reported the problem.

>status P15

The last thing you did to P15 was to attach it to ACK14.
The next thing you can do to P15 is to dispatch ACK14 to BARRETT@USER.

XCP knows where the protocol for P15 was last interrupted, and that ACK14 is related to P15 and must be handled next as part of the working-out of the protocol for P15.

>work-on ACK14 — *Resume the protocol for ACK14.*

Now you can dispatch ACK14 to BARRETT@USER.
First, you can enter text to be sent.

Your options are: text <filename> ^Z ?
Your choice? stdack.txt

XCP continues to ask for more text until Winter signals that there is no more text to be input by typing control-Z.

Your options are: text <filename> ^Z ?
Your choice? ^Z

Do you want to dispatch ACK14 to BARRETT@USER now?
Your options are: yes no ^Z ?
Your choice? yes

ACK14 has been dispatched to BARRETT@USER.
You have completed your work on ACK14.

Winter dispatches the acknowledgement to the actor who originated the problem report. If Winter had replied "no" or "control-Z" to the dispatch question, the protocol would have been suspended at that point until Winter resumed it by giving XCP the command to work-on ACK14. Winter is told that all work has been completed on ACK14. This means that ACK14 or any objects created as a result of ACK14 will never return to the WORK-ASSIGNER role.

Now you can dispatch P15 to someone in DEVELOPER.
First, you can enter text to be sent.

Recall that this protocol required an acknowledgement to be sent to the originating USER before the problem is forwarded to a DEVELOPER. Since that obligation has been fulfilled, the protocol now continues by prompting WINTER to assign the problem to a DEVELOPER.

Your options are: text <filename> ^Z ?
Your choice? t
Type your text and end it with control-Z.
This is a serious problem and should be fixed as soon as possible.^Z

Your options are: text <filename> ^Z ?
Your choice? ^Z

Next step is to dispatch P15 to someone in DEVELOPER.
Your options are: <specific person> anyone ^Z ?
Your choice? gordon

Winter sends it to the actor GORDON in the role of DEVELOPER. Had it been sent to the role but to no specific person, presumably someone in the role would take charge of it.

P15 has been dispatched to GORDON@DEVELOPER.

Winter has done all work possible on both ACK14 and P15. GORDON@DEVELOPER will eventually produce a fix report for P15 for which Winter must send out a user reply to the originating user BARRETT@USER. Now Winter can ask XCP to work on some other object, for status information, to assume another role, or to log out.

>status P15

The last thing you did to P15 was to dispatch it to GORDON@DEVELOPER.
You are waiting for a reply to arrive from GORDON@DEVELOPER.

>logout
Leaving XCP.

5. Status of the XCP Project

An *XCP* prototype has been implemented, as have several application protocols. Each protocol is layered on top of the general coordinator capabilities, and supports organizational activity by

executing the plan of cooperative activity. The application area of immediate interest is problem reporting, and the first protocols written have been in this domain. However, there is nothing in the general coordinator capabilities which ties *XCP* solely to problem reporting, and we hope to extend some of these protocols to encompass more comprehensive software project management procedures.

The first application to which *XCP* is being applied is to manage problem reporting for two Digital Equipment Corporation expert systems. XCON (also called R1) is the expert configurer [McDermott82a], and XSEL is the expert sales assistant [McDermott82b]. An XCON/XSEL Problem-Reporting Protocol has been implemented. When layered on top of *XCP*, this protocol allows the people who report, filter, assign responsibility for, and fix XSEL/XCON problems to carry out their role-defined responsiblities. Its users will include the salespeople who use XSEL to help configure customer orders, and the developers and component database engineers who support the XSEL and XCON users.

Another application to which *XCP* is being applied is to manage problem reporting for the *XCP* project itself. An *XCP* Problem-Reporting Protocol (XPRP) has been implemented, which tracks the flow of problem reports and related documents among the *XCP* developers and users.

XCP has a simplistic crash recovery mechanism which is satisfactory for the prototype. The PT is checkpointed from time to time by copying its state to a file. In the event of a crash, this file can be used to restart the PT. No effort has been made to keep track of changes made between checkpoints.

A database management system (DBMS) for *XCP* is planned for three reasons. First, the commands which provide the user with information about the state of objects could then be implemented so that the CI directly accessed the DBMS rather than requesting the PT to do so. This would result in less message traffic and faster service to the user. Second, it would be possible to implement a true transaction system so that little or no user work would be lost in a crash. Third, the prototype has a knowledge representation which uses dynamic memory to hold all persons, roles, actors, and objects known to the system. As these increase, the amount of dynamic memory available steadily decreases, performance degrades, and eventually a point would be reached where there would be no free dynamic space and Lisp would spend all of its time garbage collecting. A DBMS would use disk storage and avoid this problem.

An important scenario we envision is that there will be different *XCP* applications on various computers in a network. One of those applications will be the *XCP* project itself running the XPRP. Each application will want to be able to report problems to the *XCP* developers automatically. To accommodate this, a method of transparent communication between the XPRP and the satellite applications has been designed and is being implemented, using the Message Router product to access the DECnet™ protocols.

To simplify the task of the protocol designer/implementor, we plan further research into the problem of helping a user define formal *XCP* protocols based upon informal procedures. Protocols are presently written in a procedural fashion, and it requires a considerable amount of time and expertise to write one. Developing a protocol is a difficult task; it requires the protocol designer to first capture the intentions of the people who wish to coordinate their actions, and then debug the resulting protocol. This design/debug loop may require several iterations, even for seemingly simple protocols. For these reasons, this research will focus in three areas: providing a very-high-level protocol specification language, providing an environment for quickly testing proposed protocols at the specification level (i.e., rapid prototyping of protocols), and providing a

translation mechanism from the specification into a form which *XCP* can execute.

6. Conclusions

XCP is an experimental coordinator tool which allows organizations to develop, maintain, and carry out plans of cooperative activity, called protocols. It provides a method to formalize these protocols, and can execute them. Two of its objectives are to shoulder the burden of managing the organizational complexity and to ensure that necessary communication occurs. We believe that this frees up people to use their time more productively.

An important effect is that *XCP* encourages an organization to clearly define formal procedural obligations and relationships. Thus, the benefit to an organization is magnified because not only are responsibilities clearly defined, but the transaction costs of communicating, coordinating, and deciding could be reduced through a partnership between human and machine. Another possible benefit is that *XCP* could aid new staff to learn a procedure.

Areas in which the application of *XCP* is likely to result in significant productivity gains are characterized by complex organizational procedures, frequently used procedures, procedures involving multiple organizations, or procedures in which deviation could mean a significant loss of time, effort, money, or materials. The application area in which the first protocols have been written is problem reporting. Among other possible applications are inventory management, approval cycles, documentation control, and budget/planning cycles.

Acknowledgements

Many people have provided help in various forms. We wish to acknowledge the technical contributions of Norbert McKenna and Elizabeth Augustine, who have been instrumental in the implementation of *XCP*. Elizabeth Augustine, Allan Kent, Dennis O'Connor, and especially Stanley Lee carefully read and commented on the paper. Gene Stringer, Linda Wright, Dave Stroll, Art Beane, and Dick Paciulan provided direction on the final form that the paper took. Finally, we would like to thank Dennis O'Connor for his continuing support.

References

[Cashman80] P. Cashman and A. Holt. "A Communication-Oriented Approach to Structuring the Software Maintenance Environment". ACM Software Engineering Notes, Vol. 5, No. 1 (January 1980).

[Feldman79] S. Feldman. "Make — A Program for Maintaining Computer Programs". Software Practice and Experience, Vol. 9, 255-265 (1979).

[Holt81] A. Holt and P. Cashman. "Designing Systems to Support Cooperative Activity: An Example from Software Maintenance Management". *Proceedings* IEEE Computer Society Fifth International Computer Software and Applications Conference (COMPSAC), November 1981, pp. 184-191.

[Holt83] A. Holt et al. "Coordination System Technology as the Basis for a Programming Environment". Electrical Communication, Vol. 57, No. 4 (1983), pp. 307-314.

[Kedzierski82] B. Kedzierski. "Communication and Management Support in System Development Environments". *Proceedings* Human Factors in Computer Systems (CHI '82), March 1982, pp. 163-168.

[Lampson83] B. Lampson and E. Schmidt. "Organizing Software in a Distributed Environment". *Proceedings* SIGPLAN 83 Symposium on Programming Language Issues in Software Systems. In ACM SIGPLAN Notices, Vol. 18, No. 6 (June 1983).

[McDermott82a] J. McDermott. "R1: A Rule-based Configurer of Computer Systems". Artificial Intelligence (Netherlands), Vol. 19, No. 1 (September 1982), pp. 39-88.

[McDermott82b] J. McDermott. "XSEL: A Computer Salesperson's Assistant". In Machine Intelligence 10, J. Hayes and D. Michie (editors), Ellis Horwood Ltd, 1982.

[Peters83] T. Peters and R. Waterman. In Search of Excellence: Lessons from America's Best-run Companies. Harper and Row, 1982.

[Steele83] G. Steele Jr. Common Lisp Reference Manual. Carnegie-Mellon University, 1983.