On Designing Intelligent Hypertext Systems for Information Management in Software Engineering

Pankaj K. Garg and Walt Scacchi
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782
garg or scacchi@cse.usc.edu

ABSTRACT

Information management in large scale software engineering is a challenging problem. Hypertext systems are best suited for this purpose because of the diversity in information types that is permitted in the nodes of a hypertext. The integration of a hypertext system with software engineering tools results in a software hypertext system. *We describe the design of such a system called DIF. Based on our experiences in using DIF, we recognized the need and the potential for developing a hypertext system that could utilize knowledge about its users and their software tasks and products. Such a system might then be able to act as an* active participant *in the software process, rather than being just a passive, albeit useful storage facility. As such, we define an Intelligent Software Hypertext System (*I-SHYS[1]*) as a software hypertext system which is knowledgeable about its environment and can use such knowledge to assist in the software process. This knowledge is partly embedded in the design of an* I-SHYS *(in terms of the 'agents' that* I-SHYS *supports) and partly defined during the use of* I-SHYS *(in terms of tasks that agents perform). We present a framework for defining and organizing this knowledge, describe potential uses of such knowledge, identify limits of our approach, and suggest methods for circumventing them.*

1 INTRODUCTION

Hypertext systems are useful for information management in large scale software engineering because of the diverse types of information permitted in hypertext nodes [BEH*87]. If all the information related to a software system is stored in the same hypertext then we call it a *Software Hypertext*. A Software Hypertext System which supports the management of a software hypertext can, in conjunction with various software engineering tools, provide an Integrated Software Engineering Environment [Hen86]. The advantage of this combination (hypertext system + software engineering tools) is that one can exploit the facilities of tools for automated processing of information, while facilities of the hypertext system can be used for storing and retrieving information. We have

[1]Pronounced eye-shis

designed and implemented such a software life cycle Documents Integration Facility (DIF) in the System Factory at USC [GS88].

Our experiments with DIF led us to the notion of an Intelligent Software Hypertext System (I-SHYS). DIF is a passive system with little explicit knowledge about its surrounding environment. In I-SHYS we wish to design an *active* hypertext system, which participates and assists in the process of engineering large software systems throughout their life cycle[2]. As such, a *software hypertext environment* consists of:

1. The software engineering tools that process documentable software descriptions stored as a hypertext; and

2. The software engineering tasks that people perform through it.

If we encode knowledge about the hypertext environment into the hypertext system, such that the system can actively assist in the activities of its environment, then we get an Intelligent Hypertext System. For an Intelligent Software Hypertext System (I-SHYS), we have identified three perspectives of such knowledge:

1. Knowledge of the capabilities and uses of software tools that the environment provides;

2. Knowledge about the roles people play in the software process;

3. Knowledge about the tasks and actions people perform at different stages in the software process.

Before we can formalize and encode such knowledge in an I-SHYS, we need to understand the software process from these perspectives. This paper describes our current understanding in this regard. In Section 2 we present our understanding of a Software Hypertext System by giving an overview of DIF. Issues about interfacing tools with a hypertext system are discussed in this section. Section 3 describes the software process by categorizing the roles of *agents* in the process, describing the tasks that they perform, and detailing how their tasks can be broken down into actions performed on a software hypertext. It also discusses the attributes which can be used to categorize interactions. Finally in Section 4 we summarize the discussion and suggest future work in this direction.

2 DIF: A SOFTWARE LIFE CYCLE DOCUMENTS INTEGRATION FACILITY

DIF is a software hypertext system which helps integrate and manage the documents produced and used throughout the life cycle of software projects. It was designed for use in the System Factory, an experimental laboratory created at USC to study the development, use, and maintenance of large

[2]For simplicity, we use the term "software process" as shorthand for the process of engineering large software systems throughout their life cycle.

software systems [Sca86]. It has been used in the System Factory to support the software process for more than a dozen software systems, resulting in the creation of some 40Mbytes of software hypertext[3].

DIF provides an interface to a hypertext-based information storage structure and to a structured documentation process. A hypertext of software information is built by teams of software engineers over eight life cycle activities. DIF provides several features which allow users to view information related to a software system in an integrated manner within and across projects. The document nodes are internally organized as a tree of Unix directories and files (see Figure 1). Users of DIF enter software process information into pre-defined (but redefinable) nodes of a software hypertext that are internally treated as files. Subsequently, all routine file management (e.g., creation of directories and naming of files for related document nodes) is handled by DIF. In total, the capabilities of DIF described below enable software engineers to document their software process in ways that support: (a) analysis of the consistency and completeness of formalized document nodes, (b) intra- and inter-document traceability, (c) formatting and display, (d) indexed or query-driven browsing, (e) documentation standards, (f) multi-version documents with/without sharable annotations, (g) reusable software component catalogs, and (h) online software inspections and walkthroughs.

The integration of software hypertext nodes as files and directories is invisible to the user of DIF. For instance, when entering information for the "operational requirements" of the system, the user does not have to create the file for storing the text associated with the operational requirements; this chore is automatically handled by DIF. The user need only be concerned with creating or manipulating software descriptions without being concerned about how they are stored or where. This provides an object oriented environment of persistent software system descriptions rather than simply a loose collection of files and directories. This is reflective of the 'Next Generation Operating System' envisioned by Balzer [Bal86].

DIF also allows software engineers in the System Factory to develop parts of documents in parallel without worrying about concurrent access or integration issues. Hence person A could be writing the operational requirements of the target system while person B is writing the non-operational requirements. The individual efforts are automatically merged in the same hypertext.

2.1 System Factory Structure

The organizational structure supported by DIF in the System Factory is shown in Figure 2. In the System Factory the project manager prescribes what needs to be described in each document. In turn, these prescriptions implicitly represent the software process in effect within the structure of the software hypertext. There are also potentially several projects in the factory at the same time.

[3] We have also utilized its facilities to "publish" hard-copy, laser-printed renditions of this encyclopedic software documentation, where one complete printing produced a series of listings about 4 feet tall after binding.

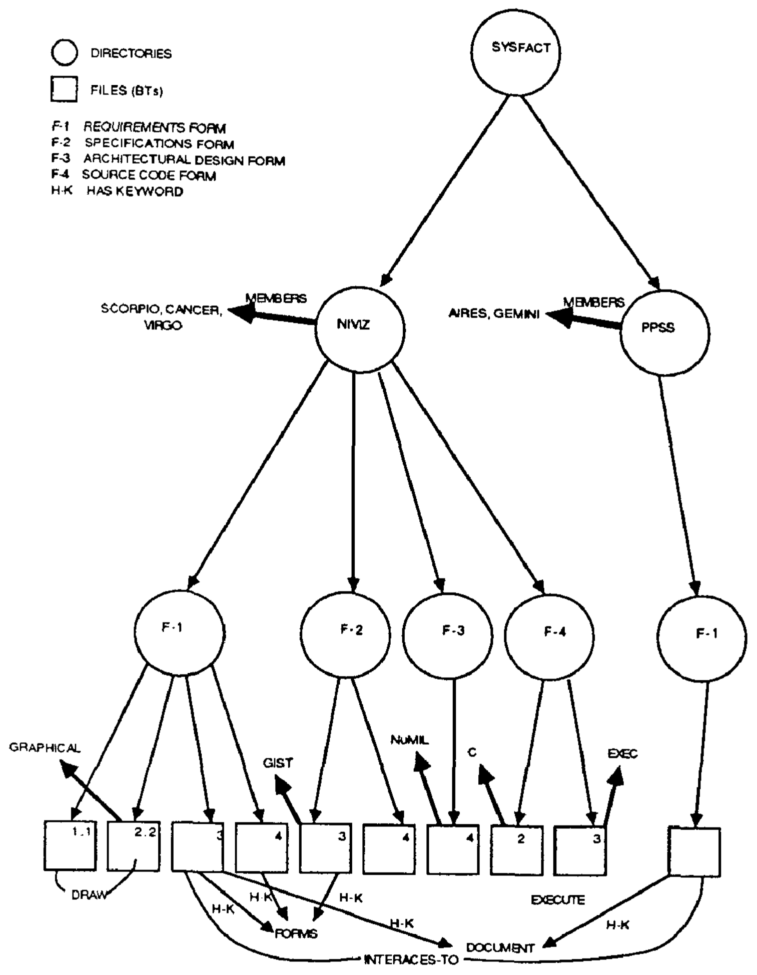


Figure 1: Hypertext of Software Documents in DIF

Several software engineers work on each project, and all projects can be constrained to follow the same software (documentation) process.

DIF supports two roles for users: a Super User role and a General User role. The two roles can be compared to the database administrator and end user respectively. In the Super User role, users define the factory structure (what projects are in the factory and who is responsible for each), and the structure of the documents (what needs to be documented). In the General User role, users exploit DIF to create, modify, and browse through the information hypertext. There are two levels at which a general user can operate: (1) at the information level, and (2) at the structure–of–the–information level. The rest of this section describes the functionalities of DIF.

2.2 Forms and Basic Templates

A Super User defines the *forms* and the *Basic Templates* (BTs) in the factory. One of the concerns of the System Factory was to ensure that all the projects have the same (standardized) structure of

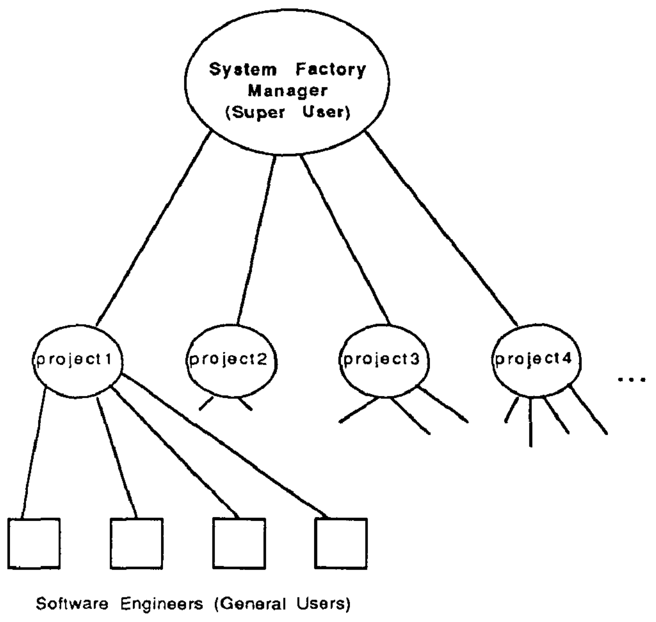

System Factory Manager (Super User)
project1
project2
project3
project4
...
Software Engineers (General Users)

Figure 2: System Factory Structure

documents. Thus each document is defined as a *form*. A form is a tree structured organization of *Basic Templates (BTs)* to be filled with information. A collection of related forms then implicitly models the software process, or tasks therein. For example, in the System Factory, the functional specifications form is shown in Table 1. Such forms provide a way of specifying the process tasks to be followed by team members working on related projects.

Section Number	Section Heading
Overview and summary of functional specification	1.0
Informal narrative specification	2.0
Narrative specification	2.1
Functional diagrams	2.2
Type lattice	2.2.1
Functional network diagrams	2.2.2
State-transition diagrams	2.2.3
Formal specification	3.0
Gist Processor Results	3.1
BT Number	BT Heading

Table 1: Functional Specifications Form

The Super User defines each form only once and all the projects inherit that form. When defining BTs the super user also defines the nature of information that needs to be given in each BT. This entails providing the attribute of the BT as being one of:

•Narrative Text	•NuMIL Structural Specifications	•C Source Code
•Graphical Diagrams	•Gist Functional Specifications	•Executable Object Code

This tells DIF which editor (e.g., a Gist language-directed editor) to use for a BT and which software tools to process the information of the BT.

2.3 Project Information

Super Users provide project information which consists of a list of projects and the software engineers working on them. This information enables DIF to check the read/write privileges of the users. General users are allowed to create, modify, and revise information related to their projects only. There is no restriction on reading the information of any project. Super Users have read/write privileges for all information.

2.4 Information Level

Facilities are provided in DIF for a user to enter, modify, and use the information required by the forms as dictated by the super user. Language-directed emacs-like editors are provided for all the formal languages that are used in the System Factory (e.g., Gist, NuMIL, and C [Sca86]). The general user enters the information in BTs without worrying about the files that need to be created. DIF automatically generates a unique filename depending on the project and the BT.

Whole forms can be stored in the revision control system, RCS [Tic82] for backup and incremental revisions. Functions supported by DIF (through RCS) include:

1. Checking in of a form. The user can ask DIF to check in a form into RCS.

2. Checking out of a form. The user can check out a whole form from RCS.

Options such as retrieving revisions through user defined identifiers, cut-off dates, etc. are available through the interface. This is an example of 'interface transparency' that DIF provides for the tools that it interfaces to (section 2.6).

Request to process the information in a BT through a software tool can be made within DIF itself without entering the operating system. For example, if a BT contains C code, the user can request its compilation. Some such requests are handled through editor interfaces [Sta84], some are built into DIF (those which require the service of a System Factory tool as opposed to a Unix tool).

Interfaces to nroff/troff, spell, etc. provide the user with a text processing environment akin to the documenters workbench [Dwb], while interfaces to mail, rn, and talk, support asynchronous and synchronous communications among project participants. Other tools available in the System Factory (e.g., application generators, computer animation environment) [Sca86] can also be interfaced with DIF in a straightforward manner.

2.5 Structure–of–Information Level

The structure–of–information[4] level allows the general user to navigate through the hypertext of information that is stored in DIF.

The user can navigate through the information in a project in the following ways:

1. **Links:** The user (super or general) can define links between BTs. The links are similar to the links allowed by most hypertext systems. Hence a person browsing the hypertext can add *annotations* to the currently visited BT, add links to other BTs, etc. For instance, the operational requirements of the system can be linked up to the code/modules that support the capability. This, therefore, provides a mechanism for maintaining consistency and traceability across documents. There are two key differences between the definition of links prevalent in hypertext literature [Con87] and those in DIF:

   - Links define relationships between existing nodes. Except for annotation links, links are precluded from creating new nodes.
   - Links are allowed to be *operational links.* This readily supports the cases where executable descriptions need to be linked to the source code. For example, a C code BT can be linked to the object code BT that represents that code. Such a link is defined in DIF as an operational link. Visiting that link results in the execution of the linked BT. Arbitrary shell procedure attachments to links will be supported in future versions of DIF.

2. **Keywords:** For each BT the user can define keywords which describe the semantics of the information contained in that BT. The decision to allow for user defined keywords rather than providing automatically generated keywords was based on the results of studies reported by researchers in information science [Sal86].

   DIF stores the keywords associated with BTs in an Ingres relation. This allows the user of DIF to use the querying facilities of Ingres for navigating through documents using keywords. For example, the user can look for all BTs (within and across projects) which have a particular keyword, list the keywords of a BT, search for BTs which have keywords satisfying a pattern, etc. Standard functionalities such as form-based querying are provided by DIF for users not trained in Quel [Que].

   An interesting feature of DIF is that it allows the reader of documents also to create keywords of their own. This allows new personnel in a project team to quickly tune the documents to their needs.

[4]This is different from a 'database schema'. In a schema the structure does not change with the information, whereas here the structure is dependent on the currently defined links.

3. **Forms and Configurations:** A Form is a tree–structured organization of BTs. This provides the user with a convenient way of viewing the documents relating to each software process activity.

   To fully utilize the potential of the hypertext of information in DIF, the user can define his/her own *configuration* of BTs. A configuration is similar to a form, except that it is not enforced on all projects but is associated with the individual user who is browsing the documents.

   Configurations can be defined, not unlike forms, by defining the constituent BTs. Configurations can also be defined on the basis of the trail which a user has followed while browsing through the information hypertext. Configurations are mainly used as a mechanism for printing hardcopy documents, much like the *path* facility suggested by Trigg [Tri83].

The user information space is restricted to the project currently being 'visited' by the user. To use the information of another project the user has to explicitly visit that project. This means that the user cannot use the information level commands on the information of projects other than the one that is being visited. Structure–level information is available regardless of which project is being visited. This is done to avoid the risk of the user getting lost in the information space [Con87]. As an additional guidance, the current project and BT are displayed in the main menu.

2.6 DIF+Tools

The basic idea in DIF is to provide a system such that all the life cycle activities can be done through DIF itself. In this sense, DIF can be considered a software engineering environment [Hen86]. With the progress of the target software system through the various life cycle activities, DIF provides a uniform interface to access the appropriate tools as necessary, e.g., a functional specification analyzer. or NuMIL Processor. In the interfaces to these tools, it supports the notion of 'interface transparency' i.e., DIF provides unobtrusive use of the tool that it interfaces to, while providing mechanisms that the tool itself lacks.

Figure 3 shows the organization of DIF with respect to the other tools in the System Factory.

The basic set of tools comprises [Sca86]:

1. A Gist Specification Analyzer and Simulator which facilitates the development and use of the formal functional specifications of software (sub)systems under development [BGW82,Sca85];

2. A Module Interconnection and Interface Definition processor that supports the design and evolution of multi-version system (module) configurations described in the NuMIL language [NS87b,NS87a];

3. An EMACS-like language-directed editing environment which helps in the construction and revision of structured documents and system description languages such as Gist, NuMIL and C; and

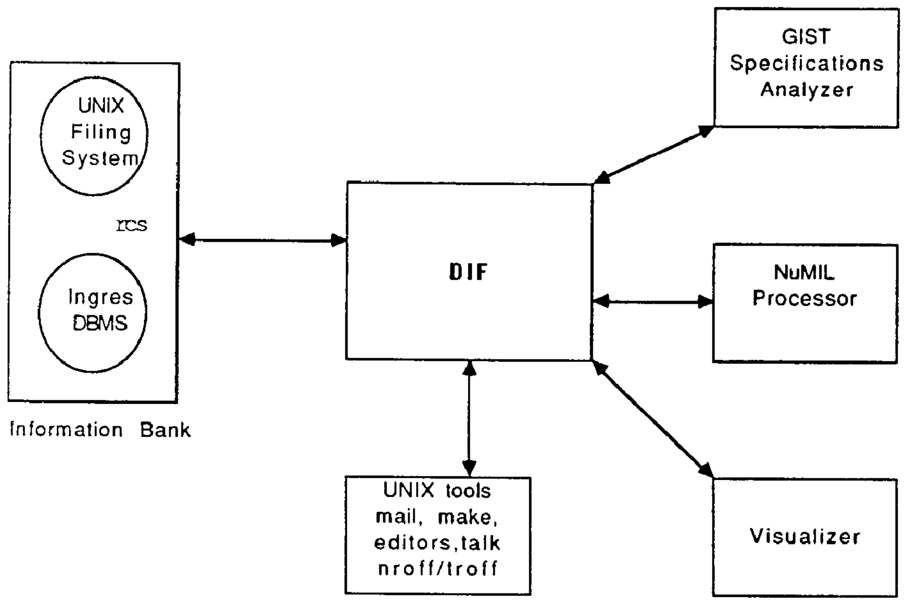


Figure 3: Organization of DIF

4. A system visualizer which graphically presents system configurations expressed in NuMIL [GS87];

5. Unix tools such as Rcs, Make, Spell, Nroff/Troff, Talk and Mail. The interface to the mailing system helps individuals to coordinate their activities by structured messages consisting of BTs.

DIF supports the notion of *Extensibility* in its interfaces to tools. It is simple to add new tools to the environment.

Tool Options: Most tools allow for the input of "switches" to tune the behavior of the tool for the application at hand. For example, while compiling a C program on Unix, the user can give a 'g' option which informs the compiler that it should generate symbol table information to be used by the symbolic debugger. An interesting way to view the switches is to consider them as means of informing the tool of some aspect of the environment in which the processing of information is taking place. For example, in the above case, the compiler is being informed that the code being compiled is an experimental one and is currently being debugged. Such information is required not only for purposes of the switch but is also useful in other places such as reporting the status of the project. The hypertext system can store this information generically, and then 'automatically' generate appropriate switches for the compiler. As another example, consider the 'c' option of the C compiler on the Unix system, which informs the compiler that the code in the file is part of a bigger system and the compiler should not load the file using a loader. This information is required at another level, viz. the architectural design of the system. Hence, the information need be given to the hypertext system only once and it can use it at multiple places. This is currently not provided by DIF and is planned for I-SHYS.

2.7 Summary

In this section we have presented an overview of a Hypertext System to manage software life cycle documents and suggested interesting ways in which smart interfaces to software tools can be provided in such a system. Our next concern is that the process model considered by DIF (described to it by way of Forms and BTs) is at a level of granularity too high to provide anything but a passive repository and processing environment that organizes document products developed through a software process. In the following section, we describe ways of breaking down the process model into finer grained actions such that they can be better supported by an I-SHYS. The treatment in this paper is semi-formal; a formal presentation is given in [Gar87b].

3 AGENT-TASK-PRODUCT PERSPECTIVE OF THE SOFTWARE PROCESS

DIF is a passive software hypertext system in that it waits for users to enter software descriptions into its network of linked nodes. However, it is incapable of interacting with its users to help elicit emerging software descriptions, nor can it explicitly represent and utilize knowledge of what roles its users play when performing different software process tasks. Instead, we would like to have a system that not only subsumes the capabilities of DIF, but does so in ways that it can eventually become *an active agent that participates in the software process.* Thus, such a hypertext system should be knowledgeable about its users, their tasks and products, and be able to ask/answer questions, simulate software process tasks, and to reason about and explain its behavior.

We first seek capture and represent knowledge about the 'agents' participating in the software process. Agents are people or intelligent systems that play well defined *roles.* An individual in the software process can play the role of more than one agent. For example, a person who has designed, implemented, and used a personal database management system is at once a designer, implementor, manager, and a user. We consider the following four categories of agents [GLB*83]:

1. *Users:* Agents who will use the target software system (end users) and/or agents who want the system developed (clients).
2. *Managers:* Agents who are responsible for software project management. There are two roles of managers that we consider, agents who coordinate the activities of other agents in the process (Process Manager, PM), and agents who analyze and evaluate the status of the process (Quality-Assurance Manager, QAM).

3. *Developers*: Agents who develop the system. Their tasks involve design and innovation, where they transform the Users' requirements into a working target system. Developer agents play roles including Analysts, Specifiers, Designers, Implementors, Testers, and Integrators.

4. *Maintainers*: Software systems are frequently modified to take care of changes in the requirements or discovery of errors in system development. Maintainers take care of such modifications, but may not have participated as the system's initial developers.

Notice that DIF considered only two categories of agents: (1) Super User (equivalent to the Manager and Client in the new framework), and (2) General User (equivalent to the End User, Maintainer, and Developers in the new framework).

The advantage of this categorization is that it helps us conceptualize the functions of agents in the process: each function viewed as a task associated with one kind of agent. This does not preclude the existence of interactions between agents, which become necessary when agents have intertwined tasks.

The next step is to identify the software process tasks of agents and to see how an I-SHYS can assist them. For this purpose we define:

- *Meta-tasks* which define the nature, configuration, and possible orderings of other tasks in the software process. Much of this is based on our study of the software process from an organizational perspective [Sca84].
- *Product tasks* that agents in the software process perform, borrowing from our empirical analyses of software engineering [Sca81,BS87] and from definitions prevalent in the software engineering literature (e.g., see [Boe81]).
- *Actions* that need to be performed in order to fulfill the commitments of tasks.
- *Primitive actions* which can be performed on a hypertext system.

The distinction between an action and a task is that a task represents the action of an agent which entails the fulfillment of some commitment [McD85]. Hence, creation of a sorting program is an action; but creation of a sorting program by an agent, A for sorting the files on a tape, T is a task.

The following sections elaborate this categorization.

3.1 Meta-Tasks

Meta-tasks are all performed by an agent assuming the role of a 'Manager'. We reiterate that an agent does not necessarily have a one-to-one mapping with the individuals in the process. The following meta-tasks have been identified [Sca84, p. 46]:

- Planning(PM[5]): Detailing the tasks that need to be performed in the software process.
- Organizing(PM): Allocating resources to the agents.
- Staffing(PM): Assigning agents to tasks.
- Directing(PM): Giving help to other agents in terms of what decisions to make in situations of uncertain or incomplete information.
- Coordinating(PM): Getting groups of people to work collaboratively.
- Scheduling(PM): Assigning time constraints on tasks.
- Validating(QAM): Confirming that the running system meets the requirements of the Users.
- Verifying(QAM): Confirming the consistency, completeness, and integrity of evolving software descriptions.

Each meta-task has a document or processable description (e.g., plans, schedules, work breakdown structure) associated with it. Accordingly, each meta-task needs to encode a different source of knowledge about the software process, and their relationship to other meta-tasks and product tasks in order to be capable of providing active participation.

3.2 Product Tasks

Product related tasks are carried out by agents assuming the role of either a Developer, a Maintainer, or a User. Users pose requirements for the system and use the system. Maintainers change the system based on emerging requirements of Users. Developers build the system using requirements posed by the Users and guidelines suggested by PMs. Several tasks can be performed in this regard:

1. Requirements Definition (Clients)
2. Requirements Analysis (Analysts)
3. Functional Specifications (Specifier)
4. Architectural Design (Designer)
5. Detailed Design (Designer)
6. Implementation (Implementor)
7. System Integration (Integrator)
8. System Delivery (Integrator)

[5] Agent category responsible for the task

9. System Use (End User)

10. Fixing bugs in the system (Maintainer)

11. Creating revisions of the system (Maintainer)

12. Enhancing the system (Maintainer)

13. Creating new versions of the system (Maintainer)

14. System bug discoveries and reporting (End User)

15. Testing (Tester)

16. Requesting Enhancements on the system (End User).

The definition of these tasks can be found in any book on software engineering (e.g., see [Boe81]). As before, each product task has an associated document or software description that is produced upon its completion. In turn, forms with computational methods attached are needed in order to help elicit the pertinent software information in order to evaluate consistency, completeness, and integrity of intra- and inter-tasks products.

The tasks viewed from this viewpoint result in a task diagram as shown in Figure 4. The figure shows the distribution of tasks of the three agents: Maintainers, Developers, and Users, with an example of a task of PM and that of a specifier elaborated. For a detailed account of the process related tasks, the reader is referred to [SBB*86]. The intertwining of tasks of multiple agents leads to requirements of interaction and collaboration, as discussed in section 3.5.

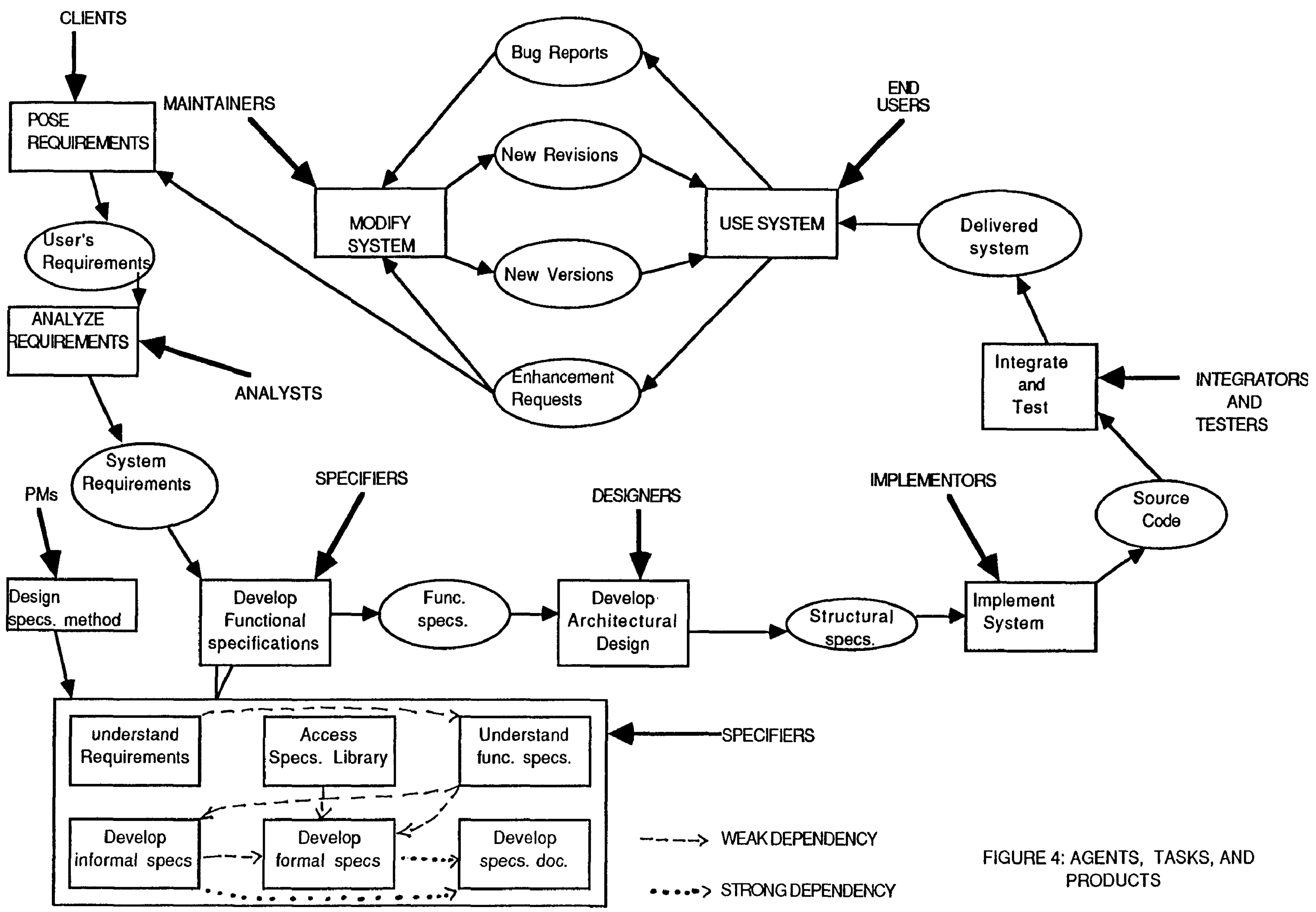


FIGURE 4: AGENTS, TASKS, AND PRODUCTS

3.3 Actions

Actions are obtained from the definition of tasks by detailing the steps that need to be done by the agent in order to effectively carry out the task.

As an example consider the `Develop functional specifications` task in Figure 4. This task can be broken down into several actions [Ben87]:

- Understand Users' requirement
- Understand functional specifications of the system
- Develop informal specifications of the system
- Access library of exemplar specifications
- Develop formal specifications of the System
- Develop a specifications document for the system.

Each action can be carried out by the Specifier agent. The dependency of actions (as shown in Figure 4) indicates that: (1) (Weak Dependency) If action A is dependent on action B, then action A cannot be effectively started unless there is a successful completion of action B. This means that A can possibly be started before the end of action B, but the results will not be as *effective* as when A is started after B is finished. For example, a specifier can start developing the formal specifications before looking at a library of exemplar specifications. (2) (Strong Dependency) If A is dependent on B, then A cannot be started unless B has finished. For example, a specifier cannot develop the specifications document unless the formal and the informal specifications have been written.

These tasks can be related to the BTs defined for the Functional Specifications Form (Table 1) as follows:

- `Understand users' requirement` leads to the informal narrative specification BT 2.0;
- `Understand functional specifications` and `Develop informal specifications` of the system lead to the Narrative Specifications BT 2.1;
- `Access library of exemplar specifications` and `Develop formal specifications of the system` lead to BTs 2.2 through 3.0; and
- `Develop specifications` document for the system leads to the Functional Specifications Form, with all its BTs integrated.

The support provided by DIF in integrating the results of multiple tasks is now clearer. But DIF deals with only the results of tasks, and not how tasks are performed. Hence it supports the tasks at a very coarse level of granularity, ignoring the actions that compose the task. In an I-SHYS our focus is to develop definitions of tasks in terms of the actions that compose them, such that the actions are simple enough to be automatically supported. As an aside, we must caution the reader that we are not suggesting to reduce the creative aspects of the process. Rather, we are suggesting to reduce its non-creative aspects which encumber creative aspects. This is in tune with the philosophy being pursued by Delisle and Schwartz in their definition of an Idea Processor [DS87], by the Colab project at XEROX, PARC [FS86] and project Nick at MCC [BCE*87].

To see how this could be achieved, consider the action Understand Users's requirements, which is part of the task of `Develop functional specifications`. It can be broken into finer grained actions as follows:

- View the requirements document;
- Create notes of the key points of the requirements;
- If something is unclear, communicate with the Users to understand it better;
- If still unclear then discuss it with fellow Developers, PMs, or Users;
- Organize notes about the understanding.

This leads to the definition of Primitive Actions on a hypertext as follows.

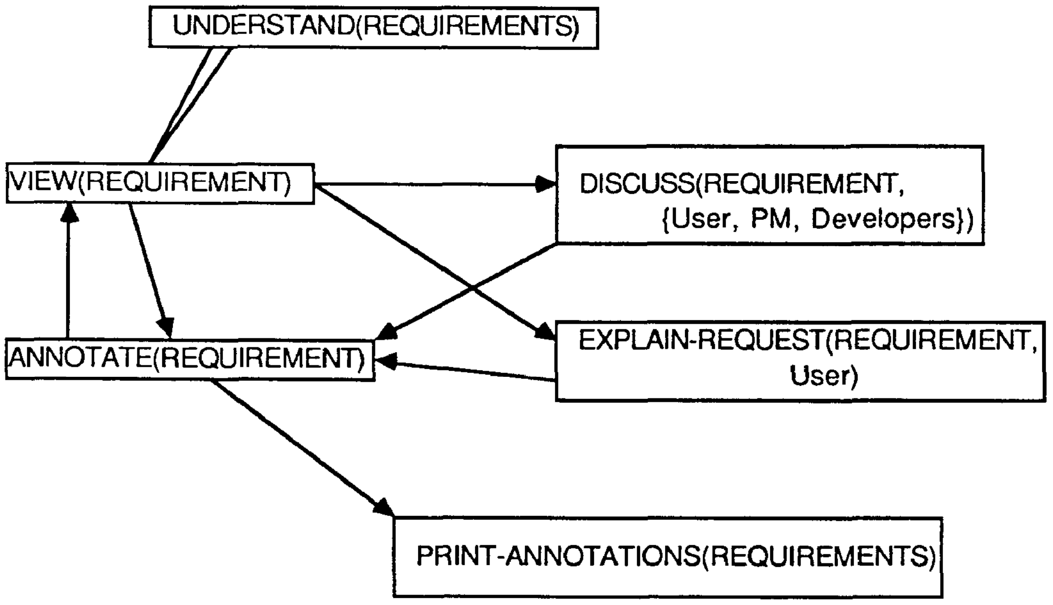
3.4 Primitive Actions

Consider the hypertext to consist of objects and relationships between the objects[6][Gar87a]. (In DIF these were BTs and links between BTs.) From the example in the previous subsection, we can identify the following primitive actions:

- VIEW(REQUIREMENT) — 'look at' an information object containing a User's requirement.
- ANNOTATE(REQUIREMENT) — attach notes of 'understanding' to an object X.
- EXPLAIN-REQUEST(REQUIREMENT, U) — request an explanation of a requirement from a User agent.
- DISCUSS(REQUIREMENT, $\{A_1, A_2, \ldots A_n\}$) — discuss a requirement with other agents of the process.
- PRINT-ANNOTATIONS(REQUIREMENTS) — organize and print the annotations attached to the requirements.

The action `Understand User's Requirements` can be understood in terms of these primitive actions as the following diagram shows:

[6]The objects are the nodes in the hypertext graph, and the relationships are the edges.


UNDERSTAND(REQUIREMENTS)
VIEW(REQUIREMENT)
DISCUSS(REQUIREMENT, {User, PM, Developers})
EXPLAIN-REQUEST(REQUIREMENT, User)
ANNOTATE(REQUIREMENT)
PRINT-ANNOTATIONS(REQUIREMENTS)

Consider the diagram as a non-deterministic flowchart. The double lined arrows show the relation 'elaboration' [SFG85] which means that the action ordering at the head of the arrow can be considered as the elaboration of the action at the tail of the arrow.

The primitive actions developed in this manner can be generalized by replacing REQUIREMENT with an arbitrary object X. For example, we can have an action UNDERSTAND(X) which is composed of VIEW(X), DISCUSS(X,α), EXPLAIN-REQUEST(X,ζ), ANNOTATE(X), and PRINT-ANNOTATIONS(X), where α is a set of agents, and ζ is the agent responsible for the creation of X.

3.5 Discussion

There are several ways that this approach can result in knowledge about software process agents, tasks, and products being encoded in an I-SHYS. In this subsection we discuss these issues.

Establishing Context: The break up of tasks into actions and of actions into primitive actions results in the establishment of *context* [DS87] of the hypertext. For example, the **Understand Users Requirements** establishes the context in which one of the five primitive actions can be taking place. An I-SHYS can therefore automatically prune out the objects and relationships that are not pertinent to one of these actions. Since the action of **Understand Users' Requirements** is carried out over a period of time, this can reduce the information overload on the agent, as the agent does not have to worry about objects and relationships which are not relevant to his/her immediate action. In current practices context establishment would have to be done manually by the users of hypertext and can lead to much semantic complexity [DS87].

Semantically Rich Commands: The analysis naturally leads to definitions of primitive actions which can be converted into semantically rich commands. For example, it is possible to provide commands such as DISCUSS(X,A) which informs the hypertext to start a discussion about object X with agent A. Coordination tools such as suggested by Winograd [Win87] and Sluzier and Cashman [SC84] can then be integrated with the hypertext system, to 'intelligently' support discussions between agents. Similarly there can be a command to PRINT-ANNOTATIONS(X) which instructs the hypertext system to organize and print the annotations attached to an object. The encoding of what kind of organization is required can be provided along the lines of *configurations* suggested in section 2.5.

Task Coordination: A break down of tasks into the actions that constitute them, allows the hypertext system to be informed about the agents and what tasks they are expected to perform. This allows the hypertext system to perform task coordination, by which it can trigger demons in case actions which were supposed to be done are not done, or if performing an action requires a response from an agent. For example, if agent A_1 performs the action EXPLAIN-REQUEST(REQUIREMENT, U_1), then agent U_1 is expected to respond with either an explanation, denial, or an alternative suggestion [Win87]. If U_1 does not respond within a certain time framework then a demon can be fired which suggests to A_1 to either: abandon the request, send a complain to a manager, or ask someone else. There are a whole set of such communication acts which arise in the course of a software process and which need automated support for their coordination [Ked83].

Interactions: As we saw in the previous examples, there are actions which explicitly require the intervention of other agents. For example, the action EXPLAIN-REQUEST(X,A) requests the explanation of object X by agent A. To support such interactions we have identified the following attributes of interactions which can possibly influence the nature of support required by them:

Agents: The categories and number of agents that interact in completing product or meta-tasks.

Communicated-object: descriptive object which is the focus of interaction.

Signal: medium or language used for the interaction. The various forms of signal of interaction that we consider are: (1) natural language, (2) figures and graphs, (3) formal language (which can be subjected to automated semantic analysis), and (4) semi-formal language (language which has a mixture of natural language, figures, graphs, and formal language).

Time: time period over which the interaction is carried out. We distinguish (1) duration of interactions as being either long or short, or (2) whether the interaction is synchronous or asynchronous.

Space: the geographical relationship of agents interacting. This can be either: (1) distributed, implying that the interacting agents are geographically dispersed and therefore cannot engage in face to face discussions; or (2) local, meaning that the interacting agents are in the same geographical location and therefore can engage in face to face discussion.

Each of the interaction attributes has an influence on the functionalities demanded of the hypertext system. Most of the functionalities can be trivially mapped from the value of the attribute. For example, a interaction carried out synchronously requires a computer mediated real time conferencing support. Similarly a interaction carried out asynchronously requires some structured mail support [MGL*87]. The advantage of encoding this information into an I-SHYS is that in a complex process such as the software process, the demands on the functionalities of the hypertext system can change depending on the stage at which the process is. If the hypertext system 'knows' about this, it can change itself to meet the requirements imposed by the stage of the process.

4 SUMMARY AND FUTURE WORK

In this paper we have described a Software Hypertext System DIF. We have then shown how notions of a Software Hypertext System can be extended to the concept of an Intelligent Software Hypertext System (I-SHYS), by studying the environment of I-SHYS from a tools, tasks, and interaction framework. Implementation of parts of I-SHYS are in progress using KnowledgeCraft on the TI Explorer LISP machine. We are also building a theoretical model of I-SHYS using an extension of the theory of Situation Calculus [McD85,Gar87b]

There is a limitation in the approach presented in this paper for the construction of I-SHYS. The limitation has been suggested by the results of empirical studies of the software process [Sca84,BS87]. An assumption made in systems such as I-SHYS is that the software process is a *closed system* wherein the tools used in the process, the tasks performed in the process, and the interaction patterns of the process, can be defined *a priori*. Empirical investigations of the process, however, indicate that the software process is an *open system* [Hew86] where the tasks and interactions are determined by complex relationships between the process, the agents in the process, the setting in which the process is carried out, and the product which is being developed. These concepts are illustrated by categorizing the tasks and interactions as:

- Primary Tasks versus Articulation Tasks [BS87,Gas86], and
- Routine Interaction versus Non-routine interaction [Gas86].

Primary tasks are the tasks prescribed by the policies of the process. Articulation tasks are a non-deterministic sequence of actions that must be performed in order to effectively carry out a primary task. Articulation tasks emerge when primary tasks descriptions are incomplete to handle unexpected circumstances such as when breakdowns, foul-ups, or resource bottlenecks suddenly emerge. An example will make this clearer[7]:

> Suppose that the primary task of an agent is to print a report detailing an understanding of the functional specifications of a system. Suppose that the agent has developed a complete understanding, and in order to print the report the use of a text-editor and a text-formatter are required. Also suppose that the agent is not familiar with any of the text editors available. Then in order to accomplish the primary task (that of printing a report) the agent has to accomplish the articulation task of understanding a text editor.

In parallel to the notions of primary tasks and articulation tasks, we distinguish between two kinds of interactions:

1. Routine interactions, and

2. Non-routine interaction.

A routine interaction occurs as part of the 'ideal' software process and not because of situations peculiar to particular settings. Programming, considered as an interaction between implementor, editor, and compiler is a routine interaction. Non-routine interactions emerge when primary software process tasks depart from expectation or plan. For example, negotiations for competing resources by multiple agents may be a non-routine interaction. Ideally a programmer should be able to follow Wirth's step-wise refinement paradigm and develop efficient 'literate programs' [Knu84]. But in real world situations, progress is not readily smooth and thus the programmer might have to try non-routine interactions (e.g., discuss issues with associates) to develop the final program.

Systems such as I-SHYS can provide support for Primary tasks and Routine interactions. For supporting Articulation tasks and Non-routine interaction, the approach suggested in this paper fails. As an alternative, one can explore the use of Case Based Reasoning. If we analyze articulation tasks and non-routine interaction, we find that these can be successfully accomplished by agents who have had similar experiences before. Case Based Reasoning provides a framework by which experiences of several agents can be encoded in a knowledge based system and the system can provide suggestions to the agents by examining the actions of agents who were faced with similar situations before. Encoding such knowledge requires a set of empirical studies which acquire knowledge about articulation tasks and non-routine interaction in various projects. It requires a framework in which to encode such knowledge such that similarity-reasoning can be performed on present situations to the situations in past cases. Our group has started efforts in this direction [Ben87,Jaz87].

[7] This example is the essence of a more detailed example presented in [BS87]

5 ACKNOWLEDGMENTS

The ideas in this paper have benefited from discussions with Salah Bendifallah and Abdulaziz Jazzar. Comments from Salah Bendifallah on earlier versions of the paper have improved the presentation. The work reported here has been supported through research grants or contracts with AT&T Information Systems, TRW Systems Engineering Design and Development, Hughes Radar Systems Group under contract number KSR576195-SN8, IBM through the Socrates project at USC, and MDA 903-81-C-0331 from DARPA to USC/ISI. In addition, the first author acknowledges the support provided by the USC graduate school through the All-University-Pre-Doctoral Merit Fellowship.

References

[Bal86] R. Balzer. Living in the Next Generation Operating System. In *Proceedings of the 10th World Computer Conference*, IFIP Congress, Dublin, Ireland, September 1986.

[BCE*87] M. Begeman, P. Cook, C. Ellis, M. Graf, G. Rein, and T. Smith. PROJECT NICK: Meetings Augmentation and Analysis. In *Computer Supported Cooperative Work Conference*, 1987.

[BEH*87] T. Biggerstaff, C. Ellis, F. Halasz, C. Kellog, C. Richter, and D. Webster. *Information Management Challenges in the Software Design Process.* Technical Report STP-039-87, MCC, Software Technology Program, January 1987.

[Ben87] S. Bendifallah. *Understanding Software Specifications Work: An Empirical Analysis.* PhD thesis, Computer Science Department, Univeristy of Southern California, December 1987. Forthcoming.

[BGW82] R. Balzer, N. Goldman, and D. Wile. Operational specification as the basis for rapid prototyping. *ACM SIGSOFT Software Engineering Notes*, 7(5):3–16, 1982.

[Boe81] B. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[BS87] S. Bendifallah and W. Scacchi. Understanding Software Maintenance Work: An Empirical Analysis. *IEEE Transactions on Software Engineering*, SE-13, March 1987.

[Con87] Jeff Conklin. Hypertext: An Introduction and Survey. *Computer*, 20(9):17–41, September 1987. Also available as MCC Technical Report no. STP-356-86, Rev. 1.

[DS87] N. Delisle and M. Schwartz. Contexts: a Partitioning Concept for Hpertexts. In *Computer Supported Cooperative Work Conference*, 1987.

[Dwb] *Documenters Work Bench.* UNIX Documentation.

[FS86] G. Foster and M. Stefik. Cognoter, theory and practice of a colab-orative tool. In *Proceedings of the Computer Supported Cooperative Work Conference*, pages 7–15, 1986.

[Gar87a] P. Garg. Abstraction Mechanisms in Hypertext. 1987. To be presented at *Hypertext '87*.

[Gar87b] P. Garg. Theoretical foundations for Intelligent Software Hypertext Systems. 1987. Computer Science Department, USC, In preparation.

[Gas86] L. Gasser. The integration of computing and routine work. *ACM Transactions on Office Information Systems*, 4(3):205–225, July 1986.

[GLB*83] C. Green, D. Luckam, R. Balzer, T. Cheatham, and C. Rich. *Report on a Knowledge Based Software Assistant.* Technical Report KES.U.83.2, Kestrel Institute, June 1983.

[GS87] P. Garg and W. Scacchi. Software Hypertext Environments for Configured Software Descriptions. 1987. Submitted for publication, 1987.

[GS88] P. Garg and W. Scacchi. A Hypertext System for Software Life Cycle Documents. January 1988. To Appear in Proceedings of the *21st Hawaii International Conference on System Sciences.*

[Hen86] P. Henderson, editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* ACM, SIGPLAN notices, vol. 22, no. 1, Palo Alto, California, Dec. 9-11, 1986.

[Hew86] C. Hewitt. Offices are open systems. *ACM Transactions on Office Information Systems*, 4(3):271–287, July 1986.

[Jaz87] A. Jazzar. *A Model for Managing Software Documents.* PhD thesis, Univeristy of Southern California, December 1987. Computer Science Department, Forthcoming.

[Ked83] B. I. Kedzierski. *Knowledge-Based Communication and Management Support in a System Development Environment.* PhD thesis, University of Southwestern Louisiana, November 1983. Avialable as Kestrel Institute Technical Report KES.U.83.3 kestrel Institute Palo Alto California.

[Knu84] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[McD85] D. McDermott. Reasoning about plans. In J. R.Hobbs and R. C. Moore, editors, *Formal Theories of the Common Sense World*, pages 269–317, Ablex Publishing Corporation, Norwood, New Jersey, 1985.

[MGL*87] T. W. Malone, K. R. Grant, K Lai, R. Rao, and D. Rosenblitt. Semi-structured messages are surprisingly useful for computer-supported coordination. In *Computer Supported Cooperative Work Conference*, 1987.

[NS87a] K. Naryanaswamy and W. Scacchi. Database Foundation to support Software Systems Evolution. *The Journal of Systems and Software*, 7(1):37–49, March 1987.

[NS87b] K. Naryanaswamy and W. Scacchi. Maintaining Configurations of Evolving Software Systems. *IEEE Transactions on Software Engineering*, SE-13(3):324–334, March 1987.

[Que] *Ingres Reference Manual.* Unix 4.2BSD Documentation.

[Sal86] G. Salton. Another look at Automatic Text-Retreival Systems. *Communications of the ACM*, 29(7):648–656, July 1986.

[SBB*86] W. Scacchi, S. Bendifallah, A. Bloch, S. Choi, P. Garg, J. Skeer, and M. J. Turner. Modelling the Software Process: A Knowledge Based Approach. 1986. System Factory Unpublished Manuscript.

[SC84] S. Sluzier and P. M. Cashman. XCP: An Experimental Tool for Supporting Office Procedures. In *IEEE 1984 Proceedings of the First International Conference on Office Automation*, IEEE Computer Society, Silver Spring MD, 1984.

[Sca81] W. Scacchi. *The Process of Innovation in Computing: A Study of the Social Dynamics of Computing.* PhD thesis, Department of Computer Science, University of California, Irvine, 1981.

[Sca84] W. Scacchi. Managing software engineering projects: a social analysis. *IEEE Trans. Software Eng.*, SE-10(1):49–59, January 1984.

[Sca85] W. Scacchi. Software specification engineering: an approach to the construction of evolving system descriptions. 1985. Research Report USC/Information Sciences Institute Marina Del Rey CA in preparation.

[Sca86] W. Scacchi. A Software Engineering Environment for the System Factory Project. In *Proc. 19th Hawaii Intern. Conf. System Sciences*, 1986.

[SFG85] A. Sathi, M. Fox, and M. Greenberg. Theory of activity representation in project management. *IEEE PAMI*, September 1985. Special issue on principles of knowledge based systems.

[Sta84] R. Stallman. EMACS: The Extensible, Customizale, Self-Documenting Display Editor. In D. R. Barstow, H. E. Shrobe, and E. Sandelwall, editors, *Interactive Programming Environments*, pages 300–325, McGraw-Hill Book Company, 1984.

[Tic82] W. Tichy. Design, Implementation, and Evaluation of a Revision Control System. In *6th International Conference on Software Engineering*, pages 58–67, Tokyo, Japan, 1982.

[Tri83] R. H. Trigg. *A Network-Based Approach to Text Handling for the Online Scientific Community.* PhD thesis, Maryland Artifitial Intelligence Group, University of Maryland, November 1983.

[Win87] T. Winograd. A language/action perspective on the design of cooperative work. In *Computer Supported Cooperative Work Conference*, 1987.