The software bookshelf

by P. J. Finnigan
R. C. Holt
I. Kalas
S. Kerr
K. Kontogiannis
H. A. Müller
J. Mylopoulos
S. G. Perelgut
M. Stanley
K. Wong

Legacy software systems are typically complex, geriatric, and difficult to change, having evolved over decades and having passed through many developers. Nevertheless, these systems are mature, heavily used, and constitute massive corporate assets. Migrating such systems to modern platforms is a significant challenge due to the loss of information over time. As a result, we embarked on a research project to design and implement an environment to support software migration. In particular, we focused on migrating legacy PL/I source code to C++, with an initial phase of looking at redocumentation strategies. Recent technologies such as reverse engineering tools and World Wide Web standards now make it possible to build tools that greatly simplify the process of redocumenting a legacy software system. In this paper we introduce the concept of a software bookshelf as a means to capture, organize, and manage information about a legacy software system. We distinguish three roles directly involved in the construction, population, and use of such a bookshelf: the builder, the librarian, and the patron. From these perspectives, we describe requirements for the bookshelf, as well as a generic architecture and a prototype implementation. We also discuss various parsing and analysis tools that were developed and integrated to assist in the recovery of useful information about a legacy system. In addition, we illustrate how a software bookshelf is populated with the information of a given software project and how the bookshelf can be used in a program-understanding scenario. Reported results are based on a pilot project that developed a prototype bookshelf for a software system consisting of approximately 300K lines of code written in a PL/I dialect.

Software systems age for many reasons. Some of these relate to the changing operating environment of a system, which renders the system ever less efficient and less reliable to operate. Other reasons concern evolving requirements, which make the system look ever less effective in the eyes of its users. Beyond these, software ages simply because no one understands it anymore. Information about a software system is routinely lost or forgotten, including its initial requirements, design rationale, and implementation history. The loss of such information causes the maintenance and continued operation of a software system to be increasingly problematic and expensive.

This loss of information over time is characteristic of legacy software systems, which are typically complex, geriatric, and difficult to change, having evolved over decades and having passed through many developers. Nevertheless, these systems are mature, heavily used, and constitute massive corporate assets. Since these systems are intertwined in the still-evolving operations of the organization, they are very

0018-8670/97/$5.00 © 1997 IBM

difficult to replace. Organizations often find that they have to re-engineer or refurbish the legacy code. The software industry faces a significant problem in migrating this old software to modern platforms, such as graphical user interfaces, object-oriented technologies, or network-centric computing environments. All the while, they need to handle the changing business processes of the organization as well as urgent concerns such as the "Year 2000 problem."

In the typical legacy software system, the accumulated documentation may be incomplete, inconsistent, outdated, or even too abundant. Before a re-engineering process can continue, the existing software needs to be documented again, or *redocumented*, with the most current details about its structure, functionality, and behavior. Also, the existing documentation needs to be found, consolidated, and reconciled. Some of these old documents may only be available in obsolete formats or hard-copy form. Other information about the software, such as design rationale, may only be found in the heads of geographically separated engineers. All of this useful information about the system needs to be recaptured and stored for use by the re-engineering staff.

As a result of these needs, we embarked on a research project to design and implement an environment to support software migration. In particular, we focused on migrating legacy PL/I source code to C++, with an initial phase of looking at redocumentation strategies and technologies. The project was conducted at the IBM Toronto Centre for Advanced Studies (CAS) with the support of the Centre for Software Engineering Research (CSER), an industry-driven program of collaborative research, development, and education, that involves leading Canadian technology companies, universities, and government agencies.

Technologies improved over the past few years now make it possible to build tools that greatly simplify the process of redocumenting a legacy software system. These technologies include reverse engineering, program understanding, and information management. With the arrival of nonproprietary World Wide Web standards and tools, it is possible to solve many problems effectively in gathering, presenting, and disseminating information. These approaches can add value by supporting information linking and structuring, providing search capabilities, unifying text and graphical presentations, and allowing easy remote access. We explore these ideas by implementing a prototype environment, called the *software bookshelf*, which captures, organizes, manages, and delivers comprehensive information about a software system, and provides an integrated suite of code analysis and visualization capabilities intended for software re-engineering and migration.

We distinguish three roles (and corresponding perspectives) involved in directly constructing, populating, and using such a bookshelf: the builder, the librarian, and the patron. A role may be performed by several persons and a person may act in more than one role. The *builder* constructs the bookshelf substrate or architecture, focusing mostly on generic, automatic mechanisms for gathering, structuring, and storing information to satisfy the needs of the librarian. The builder designs a general program-understanding schema for the underlying software repository, imposing some structure on its contents. The builder also integrates automated and semi-automated tools, such as parsers, analyzers, converters, and visualizers to allow the librarian to populate the repository from a variety of information sources.

The *librarian* populates the bookshelf repository with meaningful information specific to the software system of interest. Sources of information may include source code files and their directory structure, as well as external documentation available in electronic or paper form, such as architectural information, test data, defect logs, development history, and maintenance records. The librarian must determine what information is useful and what is not, based on the needs of the re-engineering effort. This process may be automatic and use the capabilities provided by the builder, or it may be partly manual to review and reconcile the existing software documentation for online access. The librarian may also generate new content, such as architectural views derived from discussions with the original software developers. By incorporating such application-specific domain knowledge, the librarian adds value to the information generated by the automatic tools. The librarian may further tailor the repository schema to support specific aspects of the software, such as a proprietary programming language.

The *patron* is an end user of the bookshelf content and could be a developer, manager, or anyone needing more detail to re-engineer the legacy code. Once the bookshelf repository is populated, the patron is able to browse the existing content, add annotations to highlight key issues, and create bookmarks to highlight useful details. As well, the patron can generate

new information specific to the task at hand using information stored in the repository and running the integrated code analysis and visualization tools in the bookshelf environment. From the patron's point of view, the populated bookshelf is more than either a collection of on-line documents or a computer-aided software engineering (CASE) toolset. The software bookshelf is a unified combination of both that has been customized and targeted to assist in the re-engineering effort. In addition, these capabilities are provided without replacing the favored development tools already in use by the patron.

The three roles of builder, librarian, and patron are increasingly project- and task-specific. The builder focuses on generic mechanisms that are useful across multiple application domains or re-engineering projects. The librarian focuses on generating information that is useful to a particular re-engineering effort, but across multiple patrons, thereby also lowering the effort in adopting the bookshelf in practice. The patron focuses on obtaining fast access to information relevant to the task at hand. The range of automatic and semi-automatic approaches embodied by these roles is necessary for the diverse needs of a re-engineering effort. Fully automatic techniques may not provide the project and task-specific value needed by the patrons.

In this paper we describe our research and experience with the bookshelf from the builder, librarian, and patron perspectives. As builders, we have designed a bookshelf architecture using Web technologies, and implemented an initial prototype. As librarians, we have populated a bookshelf repository with the artifacts of a legacy software system consisting of approximately 300 000 lines of code written in a PL/I dialect. As patrons, we have used this populated bookshelf environment to analyze and understand the functionality of a particular module in the code for migration purposes.

In the next section, we expand on the roles and their responsibilities and requirements. The subsequent section outlines the overall architecture of the bookshelf and details the various technologies used to implement our initial prototype. We also describe how we populated the bookshelf repository by gathering information automatically from source code and existing documentation as well as manually from interviews with the legacy system developers. A typical program-understanding scenario illustrates the use of the software bookshelf. Our research effort is also related to other work, particularly in the areas of information systems, program understanding, and software development environments. Finally, we summarize the contributions of this experience, report our conclusions, and suggest directions for future work.

Software bookshelf metaphor

Imagine an ideal scenario: where the developers of a software system have maintained a complete, consistent, and up-to-date written record of its evolution from its initial conception to its current form; where the developers have been meticulous at maintaining cross references among the various documents and application-domain concepts; and where the developers can access and update this information effectively and instantaneously. We envision our software bookshelf as an environment that can bring software engineering practices closer to this scenario, by generally offering capabilities to ease the recapture of information about a legacy system, to support continuous evolution of the information throughout the life of the system, and to allow access to this information through a widely available interface.

Our software bookshelf directly involves builder, librarian, and patron roles, with correspondingly different, but increasingly project- and task-specific, responsibilities and requirements. The roles are related in that the librarian must satisfy the needs of the patron, and the builder must satisfy the needs of the librarian (and indirectly the patron). Consequently, the builder and librarian must have more than their own requirements and perspectives in mind.

The builder. The bookshelf builder is responsible for the design and implementation of an architecture suitable to satisfy the information gathering, structuring, and storing needs of the librarian. To be relatively independent of specific re-engineering or migration projects, the builder focuses on a general conceptual model of program understanding. In particular, the schema for the underlying software repository of the bookshelf needs to represent information for the software system at several levels of abstraction.[1-3]

The levels are:

- *Physical.* The system is viewed as a collection of source code files, directory layout, build scripts, etc.
- *Program.* The system is viewed as a collection of language-independent program units, written us-

ing a particular programming paradigm. For the procedural paradigm, these units would include variables, procedures, and statements, and involve data and control flow dependencies.
- *Design*. The system is viewed as a collection of high-level, implementation-independent design components (e.g., patterns and subsystems), abstract data types (e.g., sets and graphs), and algorithms (e.g., sorting and math functions).
- *Domain*. The domain is the explanation of "what the system is about," including the underlying purpose, objectives, and requirements.

At each level of abstraction, the software system is described in terms of a different set of concepts. These descriptions are also interrelated. For instance, a design-level concept, such as a design pattern,[4] may be implemented using one or more class constructs at the program level, which correspond to several text fragments in various files at the physical level.

The builder also integrates various tools to allow the librarian to populate the bookshelf repository. Data extraction tools include parsers that operate on source code or on intermediate code generated by a compiler. File converters transform old documents into formats more suited to on-line navigation. Reverse engineering and code analysis tools are used to discover meaningful software structures at various levels of granularity. Graph visualizers provide diagrams of software structures and dependencies for easier understanding. To aid the librarian, the builder elaborates the repository schema to represent the diverse products created by these types of tools.

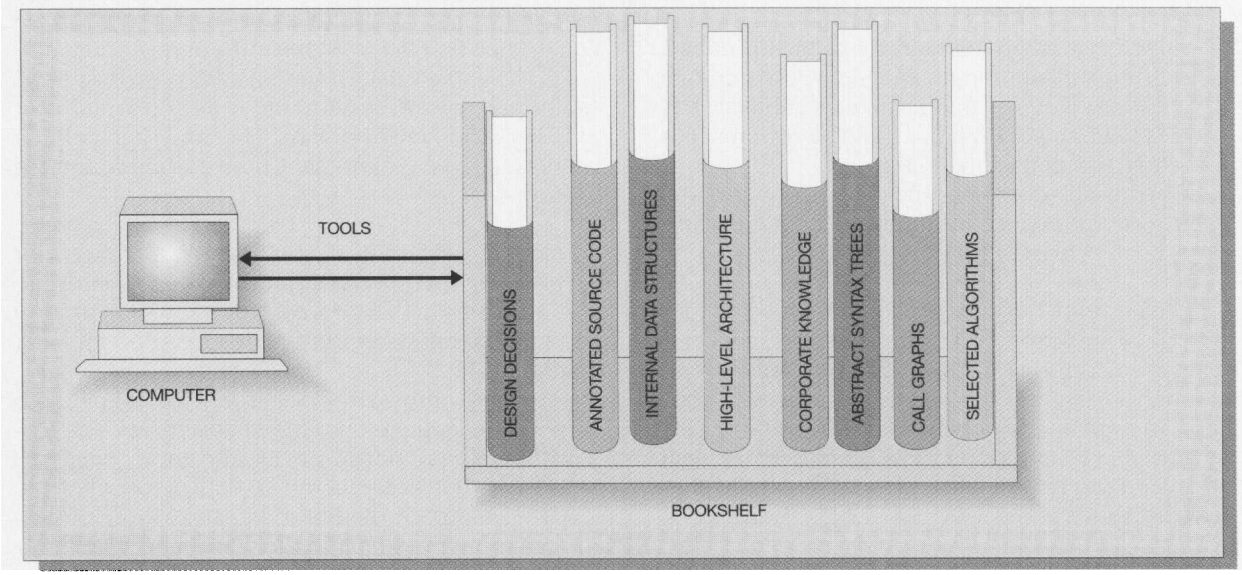
The builder has a few primary requirements. Since the information needs of the librarian and patron cannot all be foreseen, the builder requires powerful conceptual modeling and flexible information storage and access capabilities that are extensible enough to accommodate new and diverse types of content. Similarly, the builder requires generic tool integration mechanisms to allow access to other research and commercial tools. Finally, the builder requires that the implementation of the bookshelf architecture be based on standard, nonproprietary, and widely available technologies, to ensure that the bookshelf environment can be easily ported to new platforms without high costs or effort. In this paper we describe our experiences in using object-oriented database and Web technologies to satisfy these and other requirements.

The librarian. The librarian is responsible for populating the bookshelf repository with information specific to the software system. The librarian weighs the usefulness of each piece of information based on the needs of the re-engineering or migration project. The gathered information adds project-specific value and lowers the effort of the patron in adopting the bookshelf environment. The bookshelf content comes from several original, derived, and computed sources:

- *Internal*—the source code, including useful prior versions; the librarian can capture this information from the version control and configuration management system and the working development directories
- *External*—information separated from the source code, including requirements specifications, algorithm descriptions, or architectural diagrams (which often becomes out-of-date or lost when the code changes); the librarian can recover this information by talking to the developers who know salient aspects of the history of the software
- *Implicit personal*—information used by the original developers, including insights, preferences, and heuristics (which is often not verbalized or documented); the librarian can recover this information by talking to the developers and recording their comments
- *Explicit personal*—accumulated information that the developers have maintained personally, including memos, working notes, and unpublished reports (which often becomes lost when a developer leaves); the librarian can often recover this information by accessing a developer's on-line databases, along with a roadmap on what can be found
- *References*—cross-referenced information, such as all the places where a procedure is called or where a variable is mentioned (which is valuable for recovering software structure, but time-consuming and error-prone to maintain manually); the librarian can usually recover this information by using automated tools
- *Tool-generated*—diverse information produced by tools, including abstract syntax trees, call graphs, complexity metrics, test coverage results, and performance measurements (which is often not well integrated from a presentation standpoint); the librarian need not store this information in the bookshelf repository if it can be computed on demand

The librarian organizes the gathered information into a useful and easily navigable structure to the patron and forms links between associated pieces of

Figure 1 A populated software bookshelf environment



information. The librarian must also reconcile conflicting information, perhaps in old documentation, with the software system as seen by its developers. Finding both implicit and explicit personal information is critical for complementing the tool-generated content. All these difficult processes involve significant application-domain knowledge, and thus the librarian must consult with the experienced developers of the software to ensure accuracy. For the patron, the bookshelf contents will only be used if they are perceived to be accurate enough to be useful. Moreover, the bookshelf environment will only have value to the re-engineering effort if it is used. Consequently, the librarian must carefully maintain and control the bookshelf contents.

The librarian has a few primary requirements. The librarian requires tools to populate and update the bookshelf repository automatically with information for a specific software system (insofar as that is possible). These tools would reduce the time and effort of populating the repository, releasing valuable time for tasks that the librarian must do manually (such as consulting developers) or semi-automatically (such as producing architectural diagrams). The librarian requires the bookshelf environment to handle and allow uniform access to diverse types of documents, including those not traditionally recorded (e.g., electronic mail, brainstorming sessions, and interviews of customers). Finally, the librarian requires structuring and linking facilities to produce bookshelf content that is organized and easily explored. The links need to be maintained outside of the original documents (e.g., the source code) to not intrude on the owners of those documents (e.g., the developers).

The patron. The patron is an end user who directly uses the populated bookshelf environment to obtain more detail for a specific re-engineering or migration task. This role may include the developers who have been maintaining the software system and have the task of re-engineering it. Some of these patrons may already have significant experience with the system. Other patrons may be new to the project and will access the bookshelf content to aid in their understanding of the software system before accepting any re-engineering responsibilities. In any case, the patron can view the bookshelf environment as providing several entities that can be explored or accessed (see also Figure 1):

- *Books*—cohesive chunks of content, including original, derived, and computed information relevant to the software system and its application domain (e.g., source code, visual descriptions, typeset documents, business policies)

- *Notes*—annotations that the patron can attach to books or other notes (e.g., reminders, audio clips)
- *Links*—relationships within and among books and notes, which provide structure for navigation (e.g., guided tours) or which express semantic relationships (e.g., between design diagrams and source code)
- *Tools*—software tools the patron can use to search or compute task-specific information on demand
- *Indices*—maps for the bookshelf content, which are organized according to some meaningful criteria (e.g., based on the software architecture)
- *Catalogs*—hierarchically structured lists of all the available books, notes, tools, and indices
- *Bookmarks*—entry points produced by the individual patron to particularly useful and frequently visited bookshelf content

For the patron, the populated bookshelf environment provides value by unifying information and tools into an easily accessible form that has been specifically targeted to meet the needs of the re-engineering or migration project. The work of the librarian frees the patron to spend valuable time on more important task-specific concerns, such as rewriting a software module in a different language. Hence, the effort for the patron to adopt the bookshelf environment is lowered. Newcomers to the project can use the bookshelf content as a consolidated and logically organized reference of accurate, project-specific software documentation.

The patron has a few major requirements. Most importantly, the bookshelf content must pertain specifically to the re-engineering project and be accurate, well organized, and easily accessible (from possibly a different platform at a remote site). The patron also requires the bookshelf environment to be easy to use and yet flexible enough to assist in diverse re-engineering or migration tasks. Finally, the patron requires that the bookshelf environment not interfere with day-to-day activities, other than to improve the ability to retrieve useful information more easily. In particular, the patron should still be able to use tools already favored and in use today.

Building the bookshelf

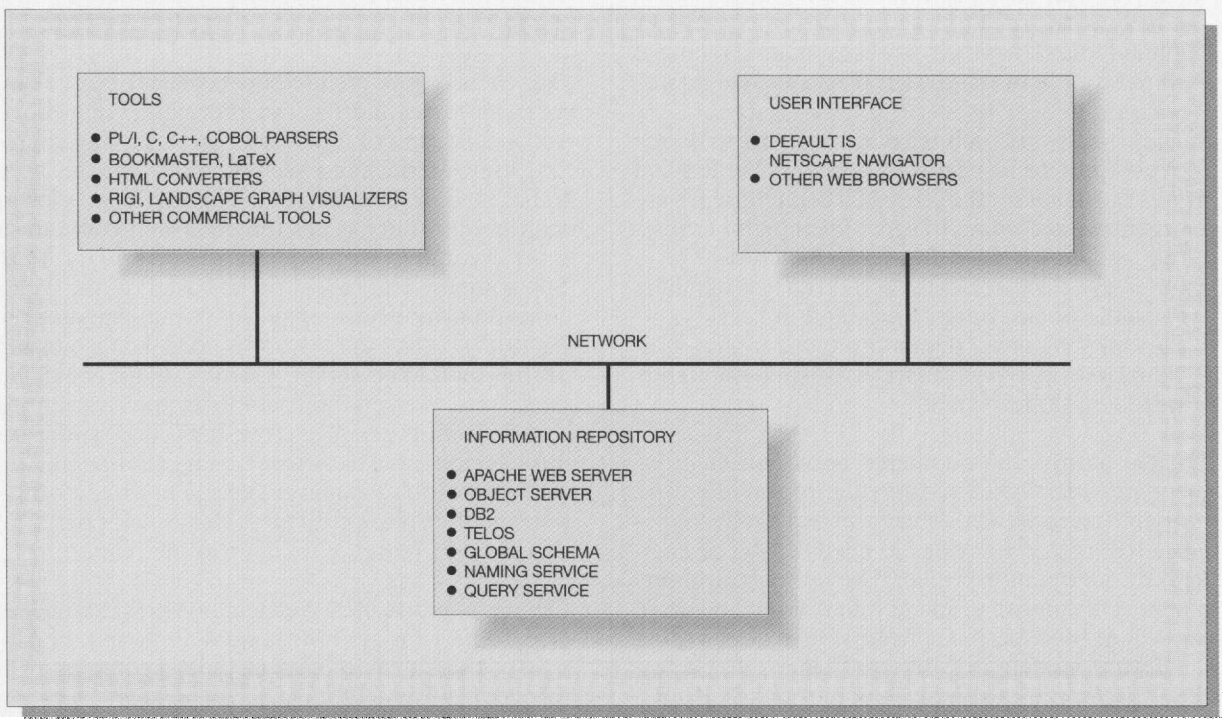
With builder, librarian, and patron requirements in mind, the builder designs and implements the architecture of the bookshelf environment to satisfy those requirements. In this section we describe our experience, from a bookshelf builder perspective, with a bookshelf architecture that we implemented as a proof-of-concept. The architecture follows the paradigm proposed in Reference 5, where a system is composed of a set of building blocks and components.[6]

Our client-server architecture consists of three major parts: a user interface, an information repository, and a collection of tools (see Figure 2). The client-side user interface is a Web browser, which is used to access bookshelf content. The server-side information repository stores the bookshelf content, or more accurately, stores pointers to diverse information sources. The repository is based on the Telos conceptual modeling language,[7] is implemented using DB2* (DATABASE 2*), and is accessed through an off-the-shelf Web server. Client-side tools include parsers to extract information from a variety of sources, scripts to collect, transform, and synthesize information, as well as reverse engineering and visualization tools to recover and summarize information about software structure. These major parts are described in more detail later in the section.

Our architecture uses Web technologies extensively (see Table 1 for acronyms and definitions). In particular, these technologies include: a common protocol (HTTP), integration mechanisms (CGI, Java**), a common hypertext format (HTML), multimedia data types (MIME), and unified access to information resources (URL). These standards provide immediate benefits by partly addressing some requirements of the builder (i.e., tool integration, nonproprietary standards, and cross-platform capabilities), the librarian (i.e., uniform access to diverse documents and linking facilities), and the patron (i.e., easy remote access). Many nonproprietary components are available off-the-shelf, including Web browsers, Web servers, document viewers, and HTML file converters, which can reduce the effort of building a software bookshelf architecture. Consequently, the use of Web technologies provides significant value to the bookshelf builder. In addition, the Web browser is easy to use and—we can assume today—immediately familiar to the patron. This lowers the startup cost and training effort of the patron in adopting the populated bookshelf environment.

User interface. The patron navigates through the bookshelf content using a Web browser, which may transparently invoke a variety of tools and scripts. The patron can browse through books or notes by simply *clicking* (a selection using a mouse button) on any links. We implemented a hypermedia link mechanism to support relationships between vari-

Figure 2 Builder perspective of the implemented bookshelf architecture



ous pieces of content. This mechanism allows the librarian to provide the patron a choice among possible destinations. For instance, clicking on a procedure name in a source code file may present a list of options, including the definition of the procedure in source code, its interface declaration, its position within a global call graph, the program locations that call it, and its internal data and control flow graphs. Once the patron chooses an option, a particular view of the procedure can be presented by the browser or by one of the integrated presentation tools in the bookshelf environment. This *multiheaded* link mechanism thus offers the librarian added flexibility in organizing and presenting access to bookshelf content.

We chose Netscape Navigator** as the default Web browser for the bookshelf environment, but any comparable browser should suffice. The browser must, however, support Java[8] directly since this is used as a client-side extension mechanism. In particular, this mechanism enables any browser that connects to the Web server to be transparently extended to handle various data objects in the information repository. Navigator also supports remote invocation features to allow tools to tell it to follow a URL. In following the URL, Navigator accesses the Web server to retrieve requested content from the information repository. For example, a tool can present a map of the bookshelf content as a graph, where clicking on a node would invoke Navigator to go to the corresponding book or note. These features also allow, for example, a code editor to request the browser to display details about a selected software artifact. This ability benefits the patron by making the bookshelf content readily and transparently accessible from the patron's development environment.

Information repository. To track all the different information sources and their cross references, the bookshelf environment contains an information repository that describes the content of the bookshelf. Access to the information repository is through a Web server. A module of this server is an object server, which is a mediator to a persistent object store. The object server and object store constitute the implementation of the repository. The structure for the stored data is specified using an object-ori-

Table 1 Web technologies

Item	Description
Common Protocol	The Web is founded on a client-server architecture. The clients and servers run independently, on different machines in different control domains. They communicate through a common protocol, the *HyperText Transfer Protocol* (HTTP). The connections are stateless; once the transaction with the client is completed, the server forgets the communication context. Version 1.1 of the HTTP protocol supports persistent connections that allow multiple transfers before the connection closes. To be served, a client issues a request to one of the servers, and the server analyzes the request, performs the requested operation (e.g., GET, POST, PUT), and generates a response.
Unified Access	The servers provide controlled access to information resources they manage. The resources are accessed by clients via links called *uniform resource locators* (URLs) that designate the location and the identity of the desired resource.
Multimedia Data Types	The data associated with requests and responses are typed using the *Multipurpose Internet Mail Extensions* (MIME). The unit of transfer between the client and the server is a MIME document, which is a typed sequence of octets.
Common Hypertext Format	The *HyperText Markup Language* (HTML) defines a composite document model. The document may contain references to other documents that are rendered in line by the client (e.g., tags, pictures, audio, video). In addition to these, the document may contain links to external documents (or parts thereof). If the type of a document is text/HTML and the document contains links to other documents, each one of these is handled in a separate transfer.
Integration Mechanism	The *Common Gateway Interface* (CGI) defines a mechanism that allows a Web server to launch and convey requests to arbitrary external programs.

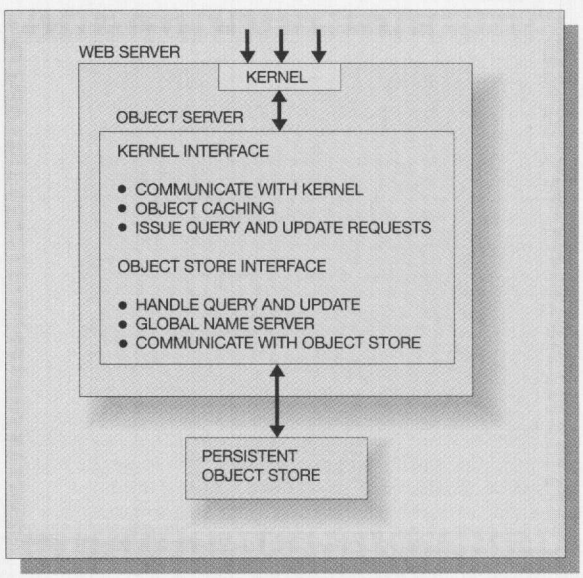
ented conceptual modeling language. By using object-oriented database technology, the bookshelf builder can provide the necessary capabilities to represent and structure highly interrelated, fine-grained data. The librarian especially needs these capabilities to express and organize software artifacts and dependencies. Furthermore, our particular choice of technology supports dynamic updates to the data schema, to allow extension to new and unforeseen types of content. Consequently, our use of object-oriented database technology provides a major benefit by satisfying some requirements of the builder (i.e., powerful conceptual modeling and extensibility to new types of content) and the librarian (i.e., structuring and linking facilities).

Meta-data repository. The information repository generally stores descriptions about pieces of bookshelf content (such as location) along with the links among the pieces. Since these descriptions constitute data about other data, they are called *meta-data*.[9] The repository explicitly stores the meta-data, not necessarily the data themselves. The actual data are left in files, databases, physical books, etc. This indirect approach is necessary since the actual data can be too large to store or too complex to fully model. Nevertheless, this detail only concerns the builder and librarian. The patron perspective is that bookshelf content is somehow delivered by the bookshelf repository.

Our repository design provides three basic capabilities for the builder and librarian: an information model, global schema, and persistent storage. The information model provides facilities for modeling the meta-data and is analogous to a data model for a database. The global schema consists of classes describing the kinds of information to be stored. This schema serves as a foundation for modeling the software implementation domain (by the builder) and modeling the application domain (by the librarian). In addition, the shared nature of this schema enables data integration for various tools. The persistent storage contains a populated instantiation of the schema.

Web server. The Web server provides an interface for accessing information in the repository. It does so by delivering the appropriate data to the requesting tool or acting directly as an information conduit. The Web server accepts HTTP requests and maps them using the repository meta-data into appropriate actions for accessing the actual content. This approach allows the server to journal all requests. The server can also cache requests, to allow specific optimizations for accessing distributed content. In our bookshelf implementation, we use the freely available

Figure 3 Bookshelf repository subsystems



Apache Web server.[10] The only additional requirement is that the server support CGI.

Object server and store. The repository is implemented by an object server and object store. The object server is an in-memory facility that offers caching, query, and naming services, built as an Apache Web server module for more efficient performance (see Figure 3). (An earlier, slower prototype used CGI and Tcl scripts to connect the Web server and repository.) The object store provides persistence for content description objects using DB2. The object server communicates with the object store through messages implemented with UNIX** sockets. The single communication channel between the object server and store ensures consistency. In addition, all queries and updates can be performed in the local workspace of the object server, thereby increasing performance. The object server can update the store according to whatever schedule is appropriate, depending on hardware availability, security issues, and usage patterns.

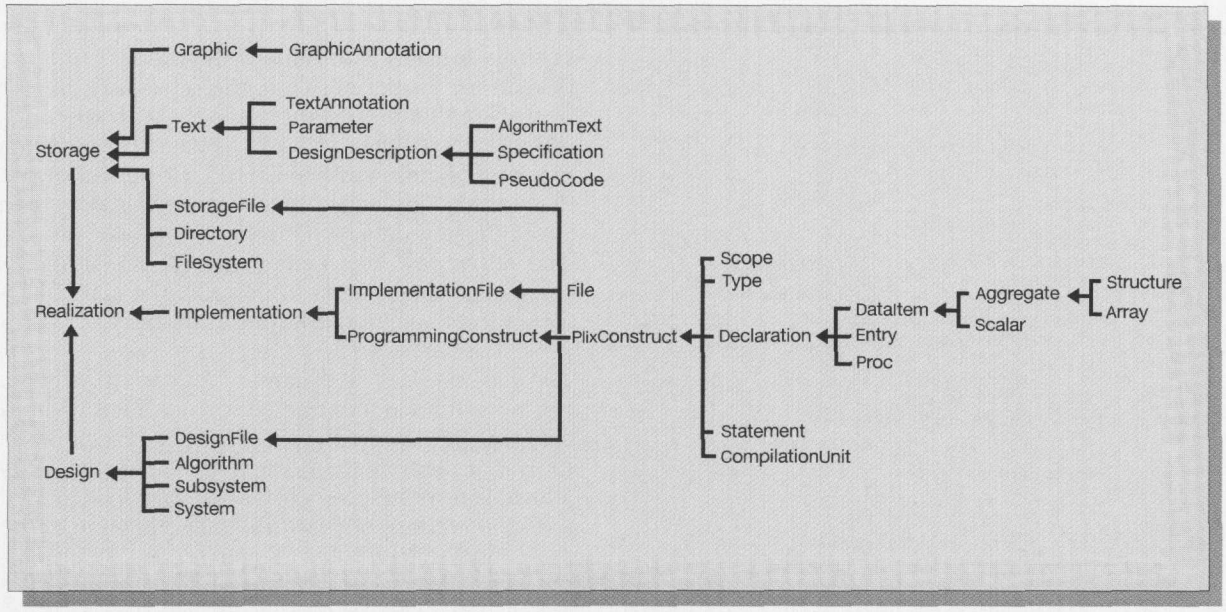
Information modeling. The information model is based on the conceptual modeling language Telos,[7] which offers facilities for organizing information in the repository using generalization, aggregation, and classification. These facilities are all necessary to satisfy the information structuring needs of the librarian. In our experience, program-understanding and re-engineering tasks require a high level of flexibility in structuring information and forming semantic associations. Telos also provides constructs for representing meta-data using metaclasses and meta-attributes. For example, links from a procedure to *called* procedures would be stored as part of the meta-description of a procedure. An interpreter/compiler for Telos is built into the object server.

Schema. The repository does not impose a predefined view of the data it is representing. Rather, a customized schema needs to be built for each application domain. This customization is a significant task and it is necessary for the builder to reduce the work of the librarian. In our customization we have tried to prepare a generally global schema that is applicable to a variety of program-understanding

Figure 4 Schema details for the Design level

```
MetaClass Design
    in DesignClass
    isa Realization
    with
    isRealizedByAttribute
      isRealizedBy : Implementation

    isPartOfAttribute
      isPartOf : Design
    hasPartsAttribute
      hasParts : Design

    isContainedInAttribute
      isContainedIn : Design
    containsAttribute
      contains : Design
end

MetaClass System
    isa Design
    ...
end

MetaClass Subsystem
    isa Design
    with
    isPartOfAttribute
      isPartOf : System
    hasPartsAttribute
      hasParts : Subsystem
    ...
end

MetaClass Algorithm
    with
    descriptionAttribute
      pseudoCode : PseudoCode
      specification : Specification
      text : AlgorithmText
    ...
end
```

Figure 5 Schema overview: the basic classes at the design, implementation, and storage levels. Nodes represent meta-classes and arcs represent is-a relations.



projects. This schema mirrors the levels of software abstraction previously outlined and includes metaclasses defining the kinds of objects that can reside in the object store. Figure 4 shows some of the design-level metaclass definitions.

According to these design-level definitions, a System is a subclass of Design and may have Subsystems as parts (with isPartOfAttribute). Design-level classes (Design and its subclasses) are realized by one or more program-level classes (Implementation and its subclasses). This is expressed by the isRealizedByAttribute of Design. For example, a specific Subsystem is a design that could be realized as a set of files. Finally, an Algorithm can be described by PseudoCode, a Specification, or in AlgorithmText.

Analogous definitions apply for the program and physical levels. Relevant metaclasses for the program level include Implementation, ProgrammingConstruct, and Statement. Similarly, Storage, FileSystem, StorageFile, and Directory are some of the classes for the physical level. Figure 5 shows these different levels of the schema.

Link mechanism. A multiheaded hypermedia link is implemented by accessing a repository object that describes possible destinations for the link. This description depends on the classes that the object instantiates. The possible destinations can be different for different types of objects (e.g., procedures versus variables) and can be further individualized for particular objects. In Telos terms, these multiheaded links are supported by multiple attributes within multiple attribute categories. For example, while browsing a procedure object, the patron may want to see different views of the procedure and its relationship to the rest of the software. By accessing the attributes in the defaultView and availableView categories, the patron can navigate to a source code view of the procedure or a text file explaining the implemented algorithm (see Figure 6).

Name translation service. The repository integrates the content found in disparate information sources. A particular procedure may be mentioned many times in different source code files and other documentation. For this procedure, there should only be a single object in the store, with appropriate attributes describing where the procedure is defined, mentioned, called, etc. Consequently, one common problem of data integration is reconciling the multiple names used for the same entity. At one extreme, a tool may have an inflexible mechanism that requires a unique name for each entity it manipulates. At the other extreme, a tool may simply manipulate the en-

Figure 6 Specific detail of the repository schema showing how attribution is used to represent hyperlinks

```
SimpleClass proc_1
  in Proc
  with
  URL
    : "http://CSER/projects/boundary.html"
  name
    : "proc_1"
  defaultView
    HTMLSourceView : proc_1_1;
  availableView
    AlgorithmView : proc_1_3;
      //algorithm
    ProcCalledByView : proc_1_2;
      //called procedures
    FullCallGraphView : proc_1_26;
      //entire call graph
    NearCallGraphView : proc_1_27;
      //neighborhood call graph
    FarCallGraphView : proc_1_28;
      //far call graph
    ProcToVarView : proc_1_29
      //accessed variables
  end
```

tities without any concern for their meaning. In addition, the implementation domain may impose restrictions on the names of entities. For example, the rules of a programming language usually require that all global procedures have unique names.

To deal with these needs, our repository provides an integrated name translation service for use by the bookshelf tools. This service is implemented by giving each entity a unique object identifier and by maintaining a mapping between this identifier and the form of name needed by a tool. This service provides additional capabilities, aside from easing data integration. In particular, this service is a basis for a general, name-based query service for use by the tools. This query service is used to support virtual links that implicitly connect entities or dynamic links that are created on demand. For example, consider a patron reading through a text document that describes the major algorithms used in a software system. This document predates the creation of the software and has almost no explicit hyperlinks. If the patron highlights a word representing the common name of an algorithm, the viewing tool could query the repository for all entities that use this name. Using the result, the tool can present the patron with a number of navigation options for further exploration of how this algorithm is implemented in the software. These navigation paths are dynamic. If it happens that these paths are useful, they can be made explicit and static, without changing the original document.

Adding new content. By design, the information repository is easily extensible with new data or types of data. In the former case, the repository creates new objects describing the new data, with appropriate pointers to the location of the data. The new data are immediately available to all tools. A tool can dynamically query the repository, fetch information about the new data, and display them to the patron. The latter case for a new type of data requires changes to the schema to describe the class of information being added to the repository.

The schema itself is dynamic. That is, the schema can be extended, contracted, or modified without changing the representation of existing objects or the tools that manipulate those objects. This flexibility allows, for example, a new type of view to be added to the procedure class without affecting any of the actual procedure instances or any of the display tools that already operate on these instances. Another use of a dynamic schema is to create user-defined views to organize and capture implicit personal information.

Tools. Our bookshelf environment is based on an open architecture, which allows a variety of tools to be integrated. Tools that populate the bookshelf repository are generally independent, communicating with each other using files and to the bookshelf Web server using standard Web protocols. These common protocols also provide the necessary integration mechanism for the Web server to export meta-data or data to external tools. These tools may use this information to locate the appropriate input files and derive new information that is stored either separately as a file, directly in the repository, or in some combination of the two. For example, a code analyzer might scan the intermediate representation of a set of program files, and calculate various complexity metrics. The results could be stored in a local file, with an entry made in the repository describing the new information and its location. In this example, the tool takes care of storing its output in a file, but updates to the repository are sent to the bookshelf Web server via Web protocols.

Adding tools. A Web browser provides only a single kind of access point into the bookshelf contents. Additional presentation tools that also access the repository are needed and should be integrated within

the bookshelf architecture using Web protocols. For example, suppose that a patron wants to edit a source code segment while also viewing an annotated version of the source in the Web browser. The patron clicks on a button on the Web page to launch the patron's favorite code editor to process the appropriate source code file. One way of implementing this feature with Web protocols is the following. The button is tied to a URL which, when clicked, causes the Web server to run a CGI script. This script encapsulates the name of the desired file using a special MIME type. The encapsulated data are sent from the server to the browser as a response. The browser recognizes these data as having a special type, and launches the appropriate helper application on the data. The helper application processes the data as a file name, consults the patron's preferences, and launches the preferred code editor to process the desired file. Such an approach relaxes the requirement for a detailed tool-modeling notation usually found in other software engineering environments.[11] In any case, a CGI script or helper application mediates between a tool and the repository, translating between the specific form required by the tool and the form required by the Web server.

Tighter integration with the bookshelf environment can be achieved by making a tool fully HTTP-aware (i.e., capable of sending and receiving HTTP requests and responses). If this is done, the tool is able to communicate with other tools and the repository more efficiently. An important step for integrating a specific tool is to describe its capabilities in terms of what kinds of views it can display (using MIME types) and what kinds of information it supplies (using the repository schema).

Dynamic content. There is a need for live, specialized, computed views as bookshelf content.[12,13] It is not possible to prefabricate all the views one might want and store them directly as static HTML pages or graphic images. There are a number of server-side solutions for creating dynamic pages. Web authors often use CGI scripts or Web server modules to construct content dynamically. Also, a metalanguage of preprocessing and transformation directives can extend HTML to provide more dynamic pages. Server Side Includes (SSI) are a primitive form of such a metalanguage.

In addition to the server-side approaches, there are also client-side strategies that operate from the Web browser, including helper applications, plug-ins, Java applets, and JavaScript** handlers. Helpers are independent programs that can provide sophisticated views. Plug-ins are software components that conform to an interface for communicating with the browser and drawing into its windows. Java applets are platform-neutral programs fetched over the network and run on a Java-enabled browser. JavaScript handlers are scripts that are triggered on certain events, such as the clicking of a link. These scripts are embedded in HTML pages and are interpreted by JavaScript-enabled browsers. All of these strategies are flexible for presenting interactive views of bookshelf data. However, some strategies may be easier to exploit than others.

To gain experience with tool integration strategies, we decided to focus on two extremes: tight and loose integration. For tight integration, the tool is essentially reimplemented in the new setting (e.g., rewritten as a Java applet). For loose integration, the tool needs to be programmable and customizable, to adapt and plug into the new setting. An annotated bibliography on different strategies for software engineering environment integration can be found in Reference 14. In the past, we had developed software visualization tools that employed graph-oriented user interfaces (i.e., Landscape,[15] Rigi,[16] and SHriMP[17]). Given our experience with these tools and the opportunity to compare these visualization techniques within the Web paradigm, we decided to integrate Landscape and Rigi into the bookshelf environment.

Integrating Landscape views. The Landscape tool[15] produces diagrams, called landscapes, of the global architecture of the target system.[18] In each diagram, there are boxes that represent software entities such as subsystems, modules, or files. Arrows connecting the boxes represent dependencies, such as calls to procedures or references to variables. These diagrams are created semi-automatically—based on software artifact information extracted automatically using parsers, together with system decomposition information collected manually from the developers through interviews. A later section in this paper illustrates how a patron uses these diagrams to obtain high-level overviews of the target software.

The original version of the Landscape tool was standalone. For the bookshelf environment, a new landscape tool was written as a Java applet. This applet displays landscape diagrams, to provide convenient navigation through the structure of the software from diagram to diagram, and to access related bookshelf content.

Integrating Rigi views. Rigi is a visualization tool for exploring and understanding large information spaces. The user interface is a graph editor that is used to browse, analyze, and modify a graph that represents the target information space. The tool is end-user programmable and extensible using the Tcl scripting language,[19] allowing user-defined views of graphs, integration with external tools, and automation of graph editing operations.[20] Also, Rigi is designed to document software architecture and to compose, manipulate, and visualize subsystem structures according to various criteria.[21]

To exploit its reverse engineering and software analysis capabilities, the Rigi tool was integrated into the bookshelf environment. The basic idea was to allow Rigi to render views constructively, based on information stored in the repository. This is an advance over approaches that only retrieve static, ready-made images. By building views dynamically, the patron can filter immaterial artifacts, emphasize relevant components, and customize the views to the analysis task at hand. The views are live and manipulable. Also, changes to the software being re-engineered are easily reflected without requiring batch updates to statically stored images.

Like Landscape, the Rigi system could be tightly integrated with the bookshelf environment by rewriting the user interface in Java. However, the programmability of Rigi allows for a loose integration strategy that requires no changes to the editor. Rigi was connected to the bookshelf environment using a CGI script and a helper application, both written in Perl.[22] Access to Rigi and its constructive views from the bookshelf Web browser had to be as simple as following a URL. Consequently, we specified a special form of URL that invokes the CGI script with a sequence of keyword/value pairs. These pairs specify required parameters, including project name, domain model, database, version, user identification, session data, display host, computational host, requested view, and context. The CGI script parses the pairs and sends the parameters to the helper application as a custom MIME type. The helper converts the parameters into Tcl and generates a custom configuration file, as well as a startup script that is used to launch Rigi to produce the view. If Rigi is already running, then the helper conveys the requested view in a file that Rigi periodically polls.

In our experience, the time needed to convey the request to Rigi is short, compared to the time needed to compute and present the requested view in a window. Since constructive views are computed by another process possibly on another machine, there are no memory problems or security limitations incurred by rendering these views within the browser using plug-in modules or Java applets. This integration strategy is generic and can be readily adapted for any stand-alone analysis tool that is end-user programmable or provides a comprehensive application program interface.

There are many strategies for integrating a tool with a Web browser. We explored two specific approaches: loose integration using CGI scripts, which allows for fast prototyping, and tight integration using Java applets, which allows for a common "look-and-feel."

The prototype brought together a diverse set of reverse engineering tools and techniques.

Parsers. The librarian requires tools to populate the bookshelf repository from existing information sources automatically, insofar as that is possible. For example, the files that belong to a software project are stored, typically, in one or more directories in a file system. The process of converting these files to HTML can be automated by "walking" the directory structure and converting the files based on their content types. Of particular interest are parsers, tools used to extract data about software artifacts and dependencies automatically. Source code files are parsed at various levels of detail according to program-understanding needs. For example, a simple parser might extract procedure calls and global variables, whereas a complete parser might generate entire abstract syntax trees.

Our use of parsers is for program-understanding purposes rather than code generation, and so the focus is primarily on extracting useful information, not all the details that would be needed for code compilation. Information useful for program understanding includes procedures (their definitions and calls) and data variables (their definitions and usage). In the implemented bookshelf environment, the parser

output is processed through further code analysis to establish links among related code fragments, compile cross references of procedures and global variables, drive visualization tools, generate architectural diagrams, produce metrics on structural complexity,[23,24] locate cloned fragments of code, and determine data and control flow paths. Since the parsers collect the locations of the extracted artifacts, the detailed analyses can be linked to the relevant fragments of code.

A simple source code parser was developed using emacs macros[25] and is currently used to parse procedure definitions and calls, and variable declarations and references. Because this parser analyzes the program source, HTML tags can be inserted in the annotated source code output at appropriate points, such as around a procedure definition. Hypertext links are generated automatically from these references using indirection (i.e., the repository maintains a mapping of references to tags), and HTML pages are generated automatically with resolved HTML tags. The parser can be extended to link the annotated code to other documentation. Similarly, external comments and notes can be attached to relevant code fragments.

A series of prototype parsers were also developed to parse two alternative program representations generated by a compiler front-end processor we were using: the cross-reference listing and the intermediate language representation. As bookshelf builders, our goal was to obtain some level of language independence by using these forms of input in some combination. In addition, parsers for these inputs are easier to write due to the limited syntax. The cross reference listing requires only a simple parser, but the reported data are selective and the format of the listing is language- and compiler-dependent. Some information is also missing, such as procedure-to-procedure calls.

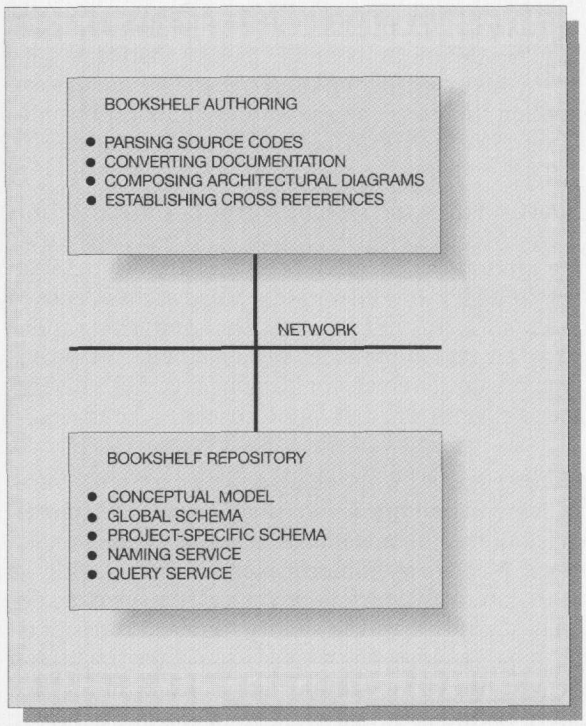
These problems can be overcome by parsing the intermediate language representation. For a family of IBM compilers, this representation is shared across multiple languages, hardware, and operating system platforms. The representation can provide detailed information to determine static control flow and, to some degree, information to calculate complexity metrics. In particular, this information includes variable type definitions, function parameter declarations, function return types, and active local and global variables. Nevertheless, in our experience, parsing only this representation is not enough since some of the information is lost. For example, the structure of file inclusions is not maintained and names of data elements generated by the front-end processor may not accurately match the variables names from the original source. Still, the approach handles the entire compiler family sharing the intermediate representation. To demonstrate this, we applied the parser to the intermediate representations of both PL/I dialect and C source code.

Shortcomings and lessons learned. The initial prototype of the bookshelf environment served as a testing ground that helped us understand where Web technologies worked well (e.g., ready access, ease of use, and consistent presentation) and where more sophisticated approaches were needed. The prototype became a vehicle for bringing together a diverse set of reverse engineering tools and techniques.

Our experience with the prototype exposed several issues with building a bookshelf using Web technologies. First, the advantage of a universally understood Web browser interface degenerates rapidly as more interactive techniques are used to give the degree of control and flexibility required for sophisticated re-engineering needs. Second, the separation between the client side and the server side introduces sharp boundaries that must be dealt with to create a seamless bookshelf environment to the patron. For example, since a client and the server run most often on different machines and file systems, there is a problem when mapping access rights between the client and server contexts. Third, the connections are stateless (as mentioned in Table 1). This creates a communication overhead when composing documents for viewing in the Web browser. Finally, no mechanism is provided for session management and version control.

The initial prototype has several limitations. First, adding a new tool required the builder to write a handcrafted CGI script, which takes some effort. Second, repository access was slow for the patron, because of the communication mechanisms used (i.e., UNIX pipes and interpreted Tcl scripts). Third, there were no security provisions to support selective access to read and possibly edit bookshelf content among patrons. Finally, maintaining a populated bookshelf repository in the face of multiple releases of the target software was another problem not addressed. Some support for multiple releases has been added to later versions of the prototype and this support is being evaluated.

Figure 7 Librarian perspective of bookshelf environment capabilities



Populating the bookshelf

With patron requirements in mind, the librarian populates the bookshelf repository with project-specific content to suit the needs of the re-engineering or migration effort. In this section, from a librarian perspective (see Figure 7), we describe our experience in populating the initial bookshelf prototype with a target software system. This target software is a legacy system that has evolved over twelve years, contains approximately 300K lines of highly optimized code written in a dialect of PL/I, and has an experienced development team. This system is the code optimization component of a family of compilers. In this paper, the name used to refer to this system is SIDOI.

Gathering information manually. As with many legacy systems, important documentation for SIDOI existed only in hard-copy versions that were filed at the back of some developer's shelf. One major need was to discover these documents and transform them into an electronic form for the software bookshelf. Consequently, over a one-year period, the members of the bookshelf project interviewed members of the development team, developed tools to extract software artifacts and synthesize knowledge, collected relevant documentation, and converted documents to more usable formats.

Most of the information for the bookshelf repository was gathered or derived automatically from existing on-line information sources, such as source code, design documents, and documentation. However, some of the most valuable content was produced by interviewing members of the development team.

Recovering architectures. The initial view of the legacy system was centered around an informal diagram drawn by one of the early developers. This diagram showed how the legacy system interfaced with roughly 20 other major software systems. We refined this architectural diagram and collected short descriptions of the functions of each of these software systems. The resulting diagram documented the external architecture of the legacy system. At roughly the same time, the chief architect was interviewed, resulting in several additional informal diagrams that documented the high-level, conceptual architecture (i.e., the system as conceived by its owners). Each of these diagrams was drawn formally as a software landscape.

The first of these diagrams was simple, showing the legacy system as composed of three major subsystems that are responsible for the three phases of the overall computation. The diagram also showed that there are service routines to support these three phases, and that the data structure is globally accessed and shared by all three phases. There were also more detailed diagrams showing the nested subsystems within each of the major phases. Using these diagrams, with a preliminary description of the various phases and subsystems, we extracted a terse but useful set of hierarchical views of the abstract architecture.

After some exploration with trying to extract the concrete architecture (i.e., the physical file and directory structure of the source code), we found it more effective to work bottom-up, collecting files into subsystems, and collecting subsystems into phases, reflecting closely the abstract architecture. This exercise was difficult. For example, file-naming conventions could not always be used to collect files into subsystems; roughly 35 percent of the files could not be classified. The developers were consulted to determine a set of concrete subsystems that included

nearly all of the files. The concrete architecture contained many more subsystems than the abstract architecture.

In a subsequent, ongoing experiment, we are recovering the architecture of another large system (250K lines of code). In this work we have found that the effort is much reduced by initially consulting with the architects of the system to find their detailed decomposition of the system into subsystems and those subsystems into files.

ILI data structure. The intermediate language implementation (ILI) data structure represents the program being optimized. The abstract architecture showed that understanding ILI would be fundamental to gaining a better understanding of the whole system. As a result, we interviewed the developers to get an overview of this data structure and to capture example diagrams of its substructures. This information is documented as a bookshelf book that evolved with successive feedback from the developers (see Figure 8). This book provides descriptions and diagrams of ILI substructures. The developers had repeatedly asked for a list of *frequently asked questions* about the ILI data structure, and so one was created for the book.

Effort. In addition to the initial work of extracting the architectural structure of the target system, one significant task was getting the developers to write a short overview of each subsystem. These descriptions were converted to HTML and linked with the corresponding architectural diagrams for browsing. Since there are over 70 subsystems, this work required more than an elapsed month of a developer's time. We collected relevant existing documents and linked them to the browsable concrete architecture diagrams. In some cases, such as when determining the concrete architecture, we required developers to *invent* structures and subsystem boundaries that had not previously existed. Such invention is challenging.

In our experience, the bookshelf librarian would need to acquire some application-domain expertise. In many legacy software projects, the developers are so busy implementing new features that no time or energy is left to maintain the documentation. Also, the developers often overlook parts of the software that require careful attention. Thus, the librarian must become familiar with the target software and verify new information for the bookshelf repository with the developer.

Reducing effort. We were constantly aware, while manually extracting the information, that this work is inherently time consuming and costly. We evolved our tools and approaches to maximize the value of our bookshelf environment for a given amount of manual work. It is advantageous to be selective and load only the most essential information, such as the documentation for critical system parts, while deferring the consideration of parts that are relatively stable. The bookshelf contents can be refined and improved incrementally as needed.

In a subsequent experiment with another target system, we have been able to do the initial population of its bookshelf much faster. Our support tools had matured and our experience allowed us to ignore a large number of unprofitable information extraction approaches from the first target system.

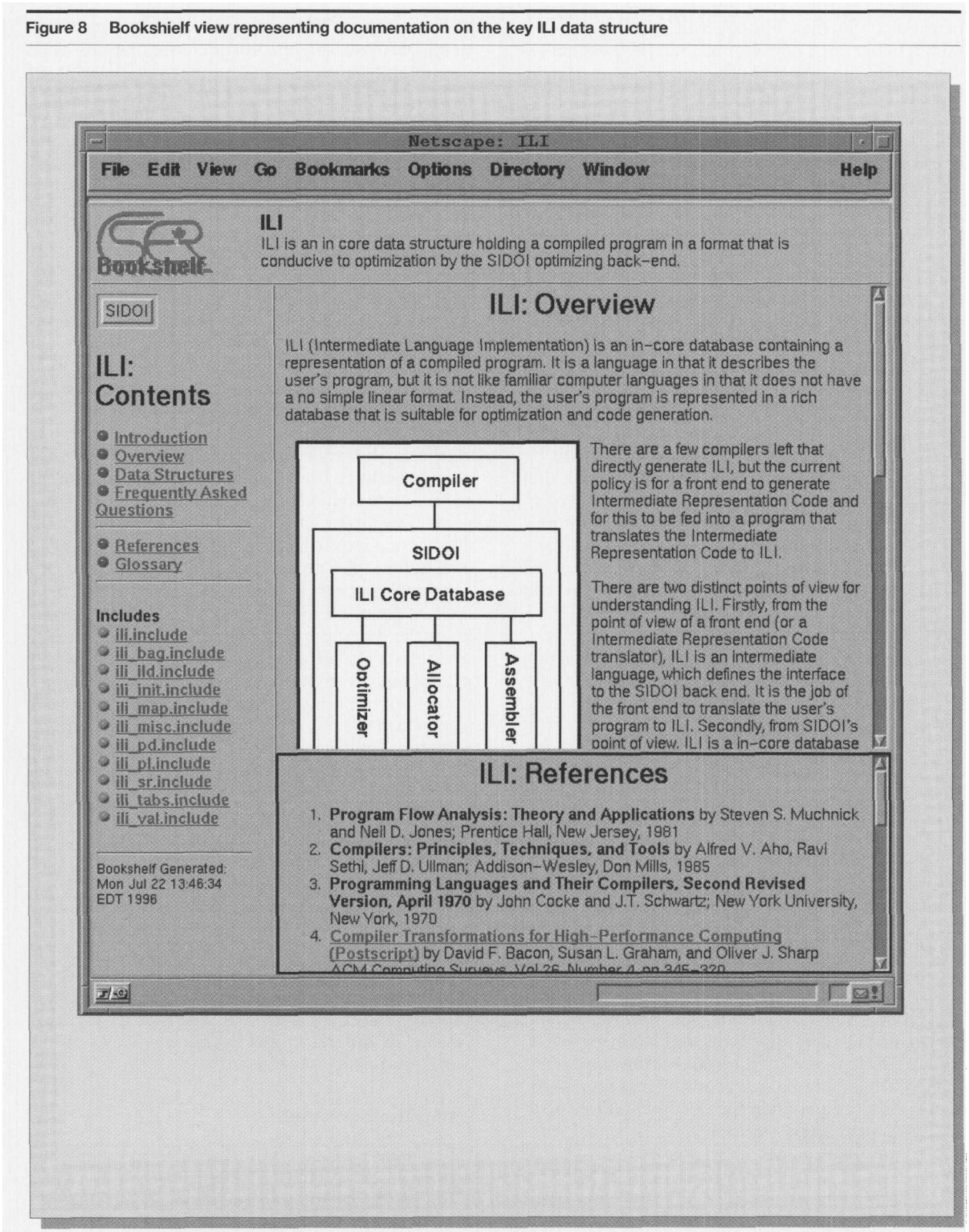
Gathering information automatically. Several software tools were used to help create and document the concrete architecture. To facilitate this effort, the parser output uses a general and simple file format. This format is called Rigi Standard Format (RSF) and consists of tuples representing software artifacts and relationships (e.g., procedure P calls procedure Q, file F includes file G). These tuple files were the basis of the diagrams of the concrete architecture. A relational calculator called Grok was developed to manipulate the tuples. To gain insights into the structure of this information, the Star system[26,27] was used to produce various diagram layouts. The diagrams were manually manipulated to provide a more aesthetic or acceptable appearance for the patrons.

Valuable information about the software was found in its version control and defect management system. In particular, it provided build-related data that were used to create an array of metrics about the build history of the project. These metrics included change frequency, a weighted defect density, and other measurements relating to the evolution of each release. A set of scripts was written that queried the version control system, parsed the responses, and gathered the desired metrics. The metrics files can be used by different tools to generate views of the evolution of the software.

Using the bookshelf

Re-engineering or migration tasks are generally goal-driven. Based on a desired goal (e.g., reducing costs, adding features, or resolving defects) and the specific task (e.g., simplifying code, increasing perfor-

Figure 8 Bookshelf view representing documentation on the key ILI data structure



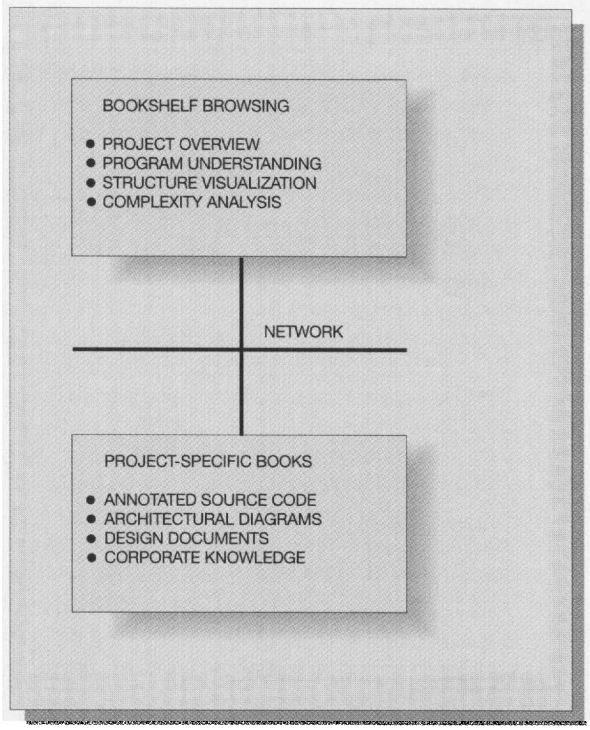
mance, or fixing a bug), the patron poses pertinent questions about the software and answers them in part by consulting the bookshelf environment data (see Figure 9). To illustrate the use of the software bookshelf, we introduce a scenario drawn from our experience with the SIDOI target system. The scenario illustrates the use of the bookshelf environment during a structural complexity analysis task by a patron who is an experienced developer.

In this scenario, the patron wishes to find complex portions of the code that can be re-engineered to decrease maintenance costs. In particular, one subsystem called DS has been difficult to understand because it is written in an unusually different style. Other past developers have been reluctant to change DS because of its apparent complexity (despite reports of suspected performance problems). Also, there may be portions of DS that can be rewritten to use routines elsewhere that serve the same or similar function. Reducing the number of such cloned or redundant routines could simplify the structure of DS and ease future maintenance work. The information gathered, while studying the complexity of DS, will help to estimate the required effort to revise the subsystem.

Obtaining an overview. The patron is unfamiliar with DS and decides to use the bookshelf environment to obtain some overview information about the subsystem, such as its purpose and high-level interactions with other subsystems. Starting at the high-level, architectural diagram of SIDOI (see Figure 10), the patron can see where DS fits into the system. This diagram was produced semi-automatically using the Landscape tool, based on the automatically generated output of various parsers. Since nested boxes express containment, the diagram (in details not shown here) indicates that DS is contained in the optimizer subsystem. For clarity, the procedure call and variable access arcs have been filtered from this diagram.

The patron can click on a subsystem box in this diagram or a link in the subsystem list in the left-hand frame to obtain information about a specific subsystem. For example, clicking on the DS subsystem link retrieves a page with a description about what DS performs, a list of what source files or modules implement DS, and a diagram of what subsystems use or are used by DS (see Figure 11). The diagram shows that DS is relatively modular and is invoked only from one or more procedures in the PL/I file optimize.pl through one or more procedures in the file ds.pl.

Figure 9 Patron perspective of a populated bookshelf environment



The page also offers links to other pages that describe the algorithms and local data structures used by DS. The algorithm description outlines three main phases. The first phase initializes a local data structure, the second phase performs a live variable analysis, and the third phase emits code where unnecessary stores to variables are eliminated. The data structure description is both textual and graphical, with "clickable" areas on the graphical image that take the patron to more detailed descriptions of a specific substructure. These descriptions are augmented by important information about the central ILI data structure of SIDOI.

After considering potential entry points into the DS subsystem, the patron decides to navigate systematically along the next level of files in the subsystem: dsinit.pl, dslvbb.pl, dslvrg.pl, and dselim.pl.

Obtaining more detail. The patron can click on a file box in the previous diagram or a file link in the list on the left-hand frame to retrieve further details about a particular source file of DS. For example,

Figure 10 High-level architectural view of the SIDOI system

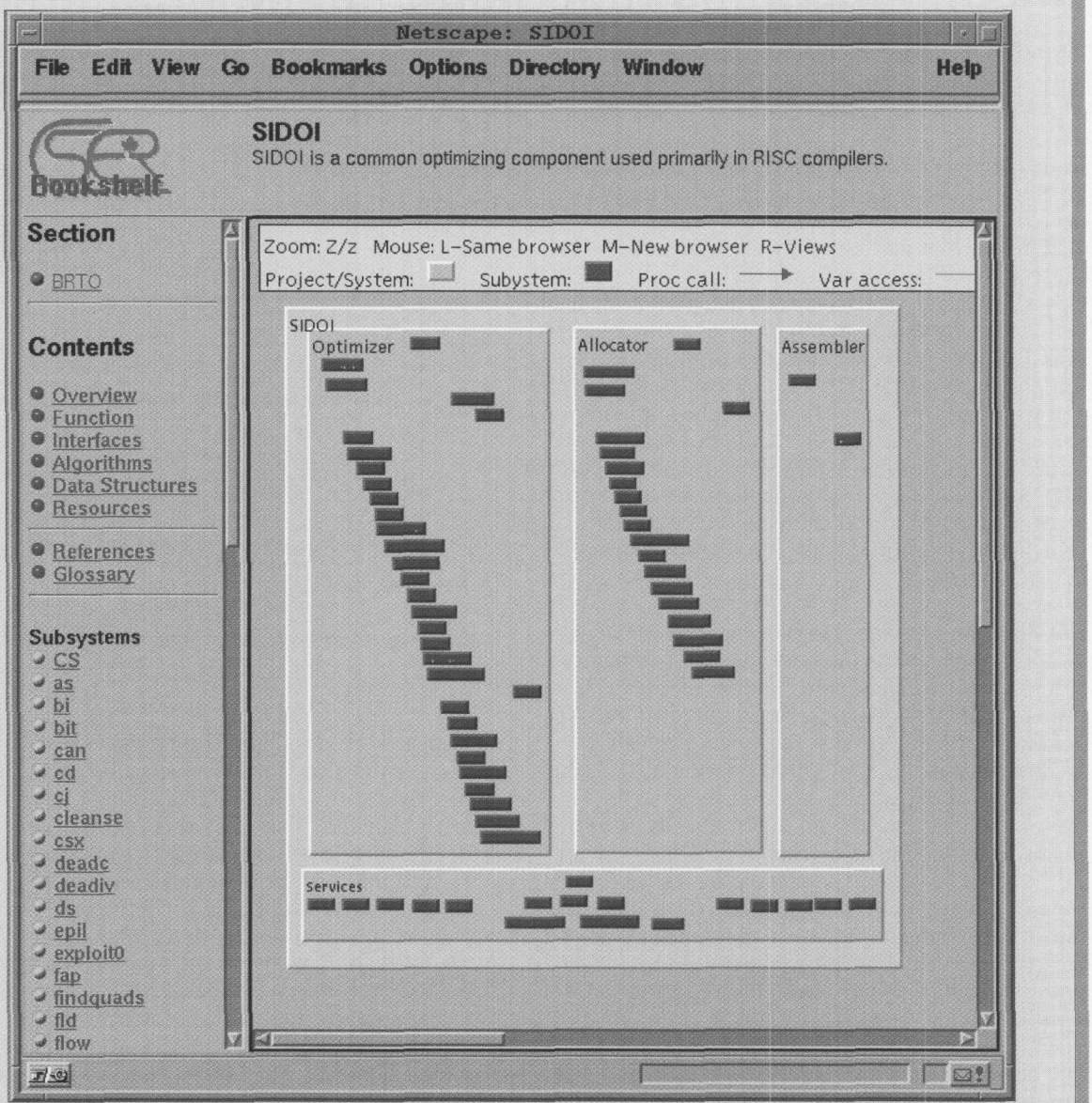


Figure 11 Architectural view of the DS subsystem

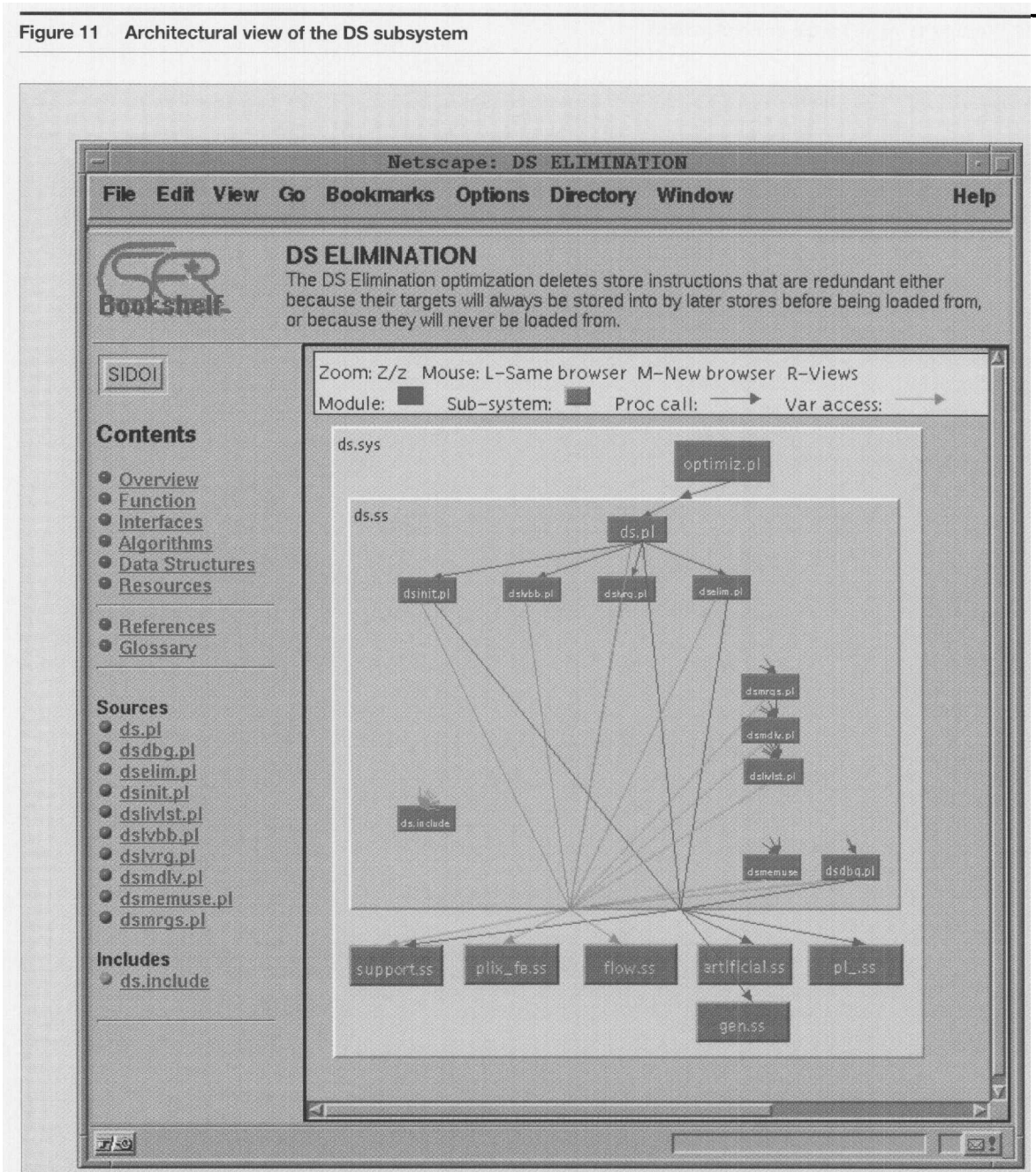
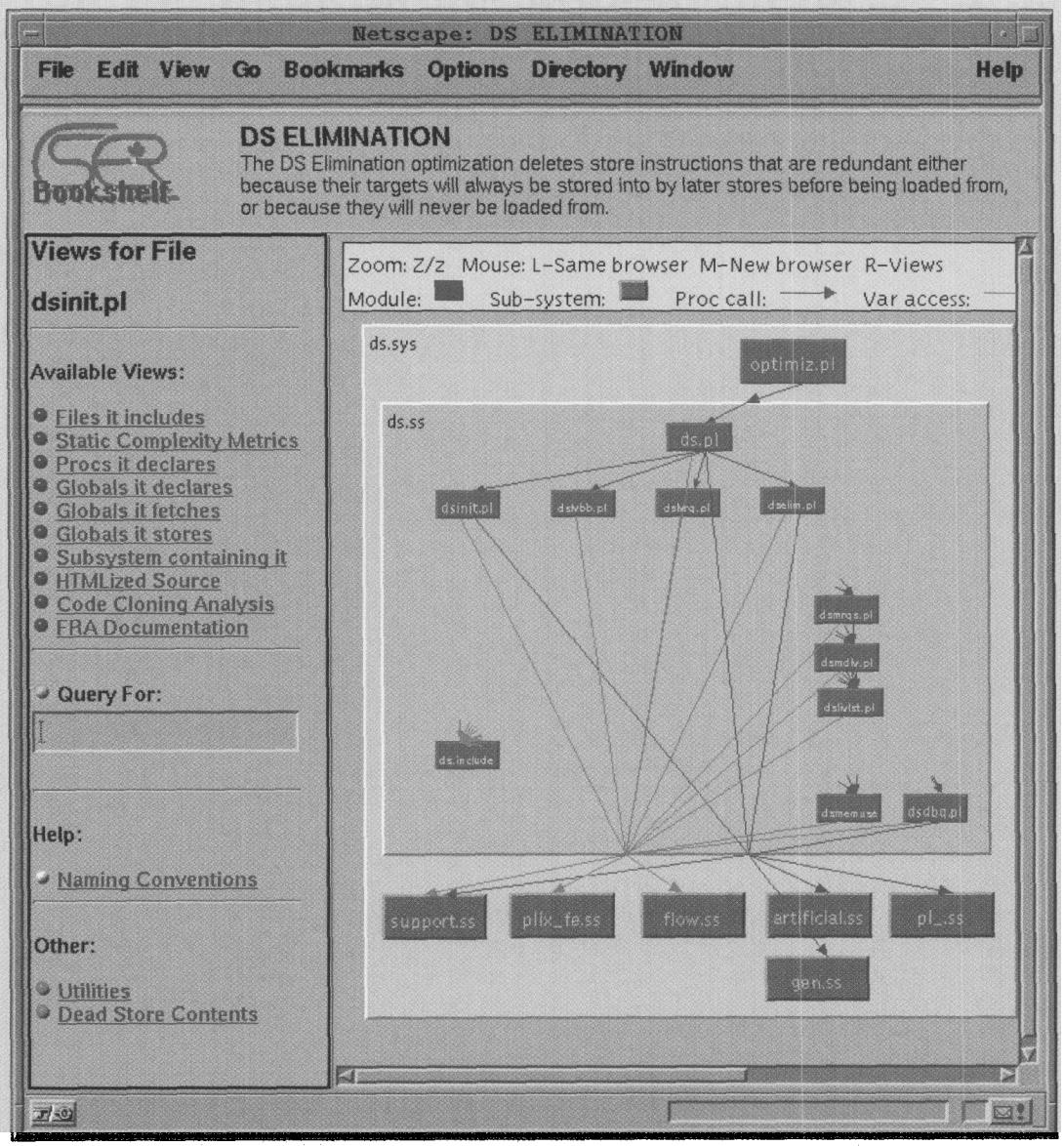


Figure 12 Available views for the dsinit.pl module



clicking on the dsinit.pl file link provides a list of the available information specific to this file and specific to the DS subsystem (see Figure 12).

The available views for a given file are outlined below.

- *Code redundancy view.* This view shows exact matches for code in the file with other parts of the system, which is useful for determining instances of cut-and-paste reuse and finding areas where common code can be factored into separate procedures.
- *Complexity metrics view.* This view shows a variety of metrics in a bar graph that compares this file with other files in the subsystem of interest.
- *Files included view.* This view provides a list of the files that are included in the file.
- *Hypertext source view.* This view provides a hypertext view of the source file with procedures, variables, and included files appearing as links.
- *Procs declared view.* This view provides a list of procedures declared in the file.
- *Vars fetched and vars stored views.* These views provide a list of variables fetched or updated in the file.

In general, clicking on a file, procedure, or variable in the diagram or set of links produces a list of the available views specific to that entity. Views appear either as lists in the left-hand frame, as diagrams in the right-hand frame, or as diagrams controlled and rendered by other tools in separate windows. Figure 13 shows a diagram generated by Rigi with the neighboring procedures of procedure dsinit. The patron can rearrange the software artifacts in the diagrams and apply suitable filters to hide cluttering information. The capabilities of the Rigi tool are fully available for handling these diagrams.

Other, more flexible navigation capabilities are provided. For instance, the patron can enter the name of a software artifact in the query entry field of the left-hand frame. This search-based approach is useful for accessing arbitrary artifacts in the system that are not directly accessible through predefined links on the current page. Also, the Web browser can be used to return to previously visited pages or to create bookmarks to particularly useful information.

Analyzing structural complexity. While focusing on the DS module, the patron decides that some procedure-specific complexity measures on the module would be useful for determining areas of complex logic or potentially difficult-to-maintain code (see Figure 14). Such static information is useful to help isolate error-prone code.[23,28–30] The bookshelf environment offers a procedure-level complexity metrics view that includes data- and control-flow-related metrics, measures of code size (i.e., number of equivalent assembly instructions, indentation levels), and fanout (i.e., number of individual procedure calls).

To examine areas of complex, intraprocedural control flow, the cyclomatic complexity metric can be used. This metric measures the number of independent paths through the control flow graph of a procedure. The patron decides to consider all the procedures in DS and compare their cyclomatic complexity values. This analysis shows that dselim, initialize, dslvbb, and dslvrg have values 75, 169, 64, and 49, respectively.

Finding redundancies. Using the code redundancy and source code views in the bookshelf environment, the patron discovers and verifies that procedures dselim and dslvbb are nearly copies of each other. Also, procedure dslvrg and dslvbb contain similar algorithmic patterns. Code segments are often cloned through textual cut-and-paste edits on the source code. Some of the clones may be worth replacing by a common routine if future maintenance can be simplified. The amount of effort needed depends on the complexity measures of the cloned code. With a pertinent set of bookshelf views, the re-engineering group can weigh the benefits and costs of implementing the revised code.

After completing the whole investigation, it is useful to store links to the discoveries in some form, such as Web browser bookmarks, guided tour books, footprints on visited pages, and analysis objects in the repository. Such historical information may help other developers with a similar investigation in the future.

Related work

In this section, we discuss related work on integrated software environments, parsing and analysis tools, software repositories, and knowledge engineering.

Integrated software environments. Tool integration encompasses three major dimensions: data (i.e., exchanging and sharing of information), control (i.e., communication and coordination of tools), and presentation (i.e., user interface metaphor).[31] Data integration is usually based on a common schema that

Figure 13 Call graph with the neighboring procedures of procedure dsinit

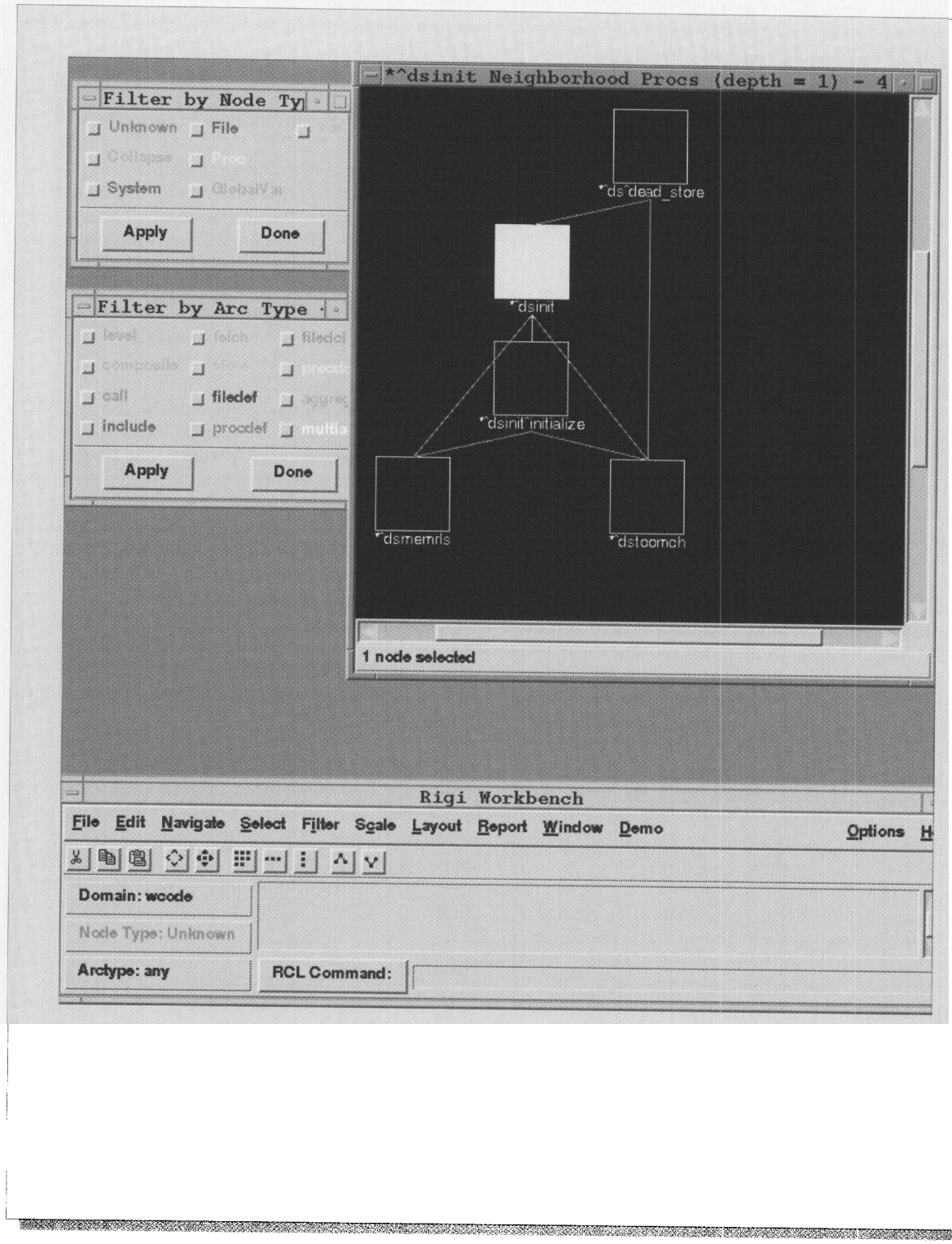
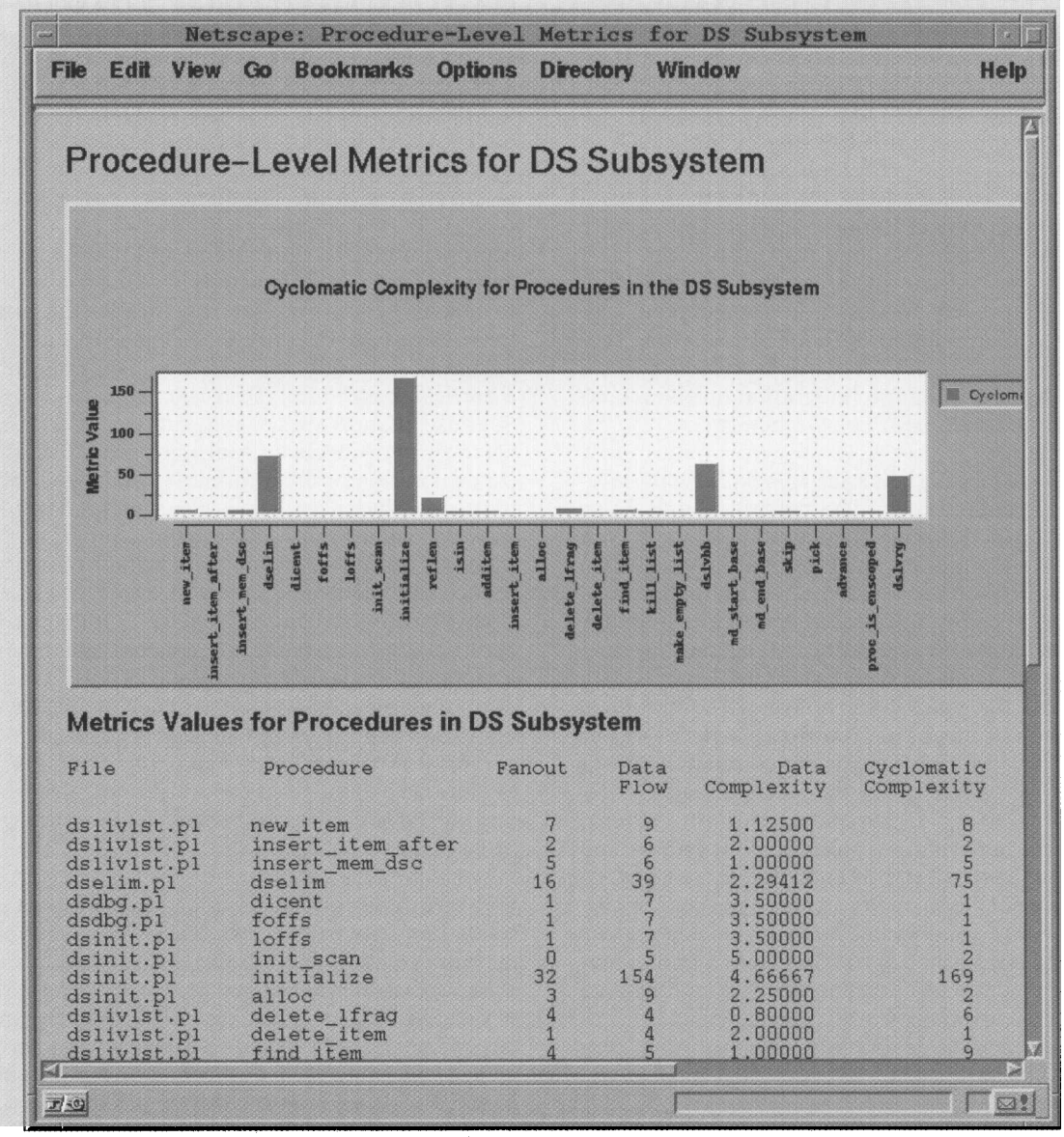


Figure 14 Procedure-specific metrics for the DS subsystem



Metrics Values for Procedures in DS Subsystem

File	Procedure	Fanout	Data Flow	Data Complexity	Cyclomatic Complexity
dslivlst.pl	new_item	7	9	1.12500	8
dslivlst.pl	insert_item_after	2	6	2.00000	2
dslivlst.pl	insert_mem_dsc	5	6	1.00000	5
dselim.pl	dselim	16	39	2.29412	75
dsdbg.pl	dicent	1	7	3.50000	1
dsdbg.pl	foffs	1	7	3.50000	1
dsinit.pl	loffs	1	7	3.50000	1
dsinit.pl	init_scan	0	5	5.00000	2
dsinit.pl	initialize	32	154	4.66667	169
dsinit.pl	alloc	3	9	2.25000	2
dslivlst.pl	delete_lfrag	4	4	0.80000	6
dslivlst.pl	delete_item	1	4	2.00000	1
dslivlst.pl	find_item	4	5	1.00000	9

models software artifacts and analysis results to be shared among different tools. For example, in the PCTE system,[32] data integration is achieved with a physically distributed and replicated object base. Forming a suitable common schema requires a study of functional aspects related to specific tool capabilities and organizational aspects in the domain of discourse. Control integration involves the mechanics of allowing different tools to cooperate and provide a common service. In environments such as Field[33] and SoftBench,[34] tools are coordinated by broadcast technology, while environments based on the Common Object Request Broker Architecture (CORBA) standard[35] use point-to-point message passing. Furthermore, control integration involves issues related to process modeling and enactment support,[36] computer-supported cooperative work,[37] cooperative information systems,[38] and distributed computing. Presentation integration involves look-and-feel and metaphor consistency issues.

The software bookshelf tries to achieve data integration through a meta-data repository and Telos conceptual model, control integration through Web protocols and scripting, and presentation integration through the Web browser hypertext metaphor. Kaiser et al. recently introduced an architecture for World Wide Web-based software engineering environments.[39] Their OzWeb system implements data integration through subweb repositories and control integration by means of groupspace services. In addition, there are several existing commercial products such as McCabe's Visual Reengineering Toolset BattleMap**,[40] which offers a variety of reverse engineering and analysis tools, visualization aids, and a meta-data repository. By and large, these environments are not based on the technologies selected for our bookshelf implementation. In particular, our work is distinguished through an open and extensible architecture, Web technology with multiheaded hypermedia links, a powerful and extensible conceptual model, and the use of off-the-shelf software components.

Parsing tools. Many parsing tools and reverse engineering environments have been developed to extract software artifacts from source files.[41] The Software Refinery[42] parses the source and populates an object repository with an abstract syntax tree that conforms to a user-specified domain model. Once populated, the user can access, analyze, and transform the tree using a full programming and query language. PCCTS is a compiler construction toolkit that can be used to develop a parser.[43] The output of this parser is an abstract syntax tree represented by C++ objects. Analysis tools can be written using a set of C++ utility functions. GENOA provides a language-independent abstract syntax tree to ease artifact extraction and analysis.[44] Lightweight parsers have emerged that can be tailored to extract selected artifacts from software systems rather than the full abstract syntax tree.[45,46] For the software bookshelf, our parsers convert the source to HTML for viewing or extract the artifacts in a language-independent way by processing the intermediate language representation emitted by the compiler front-end processor.

Analysis tools. To understand and manipulate the extracted artifacts, many tools have been developed to analyze, search, navigate, and display the vast information space effectively. Slicing tools subset the system to show only the statements that may affect a particular variable.[47] Constructive views,[48] visual queries,[49] Landscapes,[15] and end-user programmable tools[20] are effective visual approaches to customize exploration of the information space to individual needs. Several strategies have emerged to match software patterns. GRASPR recognizes program plans, such as a sorting algorithm, with a graph parsing approach that involves a library of stereotypical algorithms and data structures (*clichés*).[50] Other plan recognition approaches include concept assignment[51] and constraint-based recognition.[52] Tools have been developed for specific analyses, such as data dependencies,[53] coupling and cohesion measurements,[54] control flow properties,[41] and clone detection.[55–57] On the commercial front, several products have been introduced to analyze and visualize the architecture of large software systems.[58]

Software repositories. Modeling every aspect of a software system from source code to application domain information is a hard and elusive problem. Software repositories have been developed for a variety of specialized uses, including software development environments, CASE tools, reuse libraries, and reverse engineering systems. The information model, indexing approach, and retrieval strategies differ considerably among these uses. The knowledge-based LaSSIE system provides domain, architectural, and code views of a software system.[59] Description logic rules[60] relate the different views and the knowledge base is accessed via classification rules, graphical browsing, and a natural language interface. The Software Information Base uses a conceptual knowledge base and a flexible user interface to support software development with reuse.[61] This knowledge base is organized using Telos[7] and contains information

about requirements, design, and implementation. The knowledge base can be queried through a graphical interface to support the traversal of semantic links. The REGINA software library project builds an information system to support the reuse of commercial off-the-shelf software components.[62] Their proposed architecture also exploits Web technology.

Knowledge engineering. Related areas in knowledge engineering include knowledge sharing,[63] ontologies,[64] data repositories,[65] data warehouses,[66] and similarity-based queries.[67,68] Meta-data have received considerable attention (e.g., Reference 69) as a way to integrate disparate information sources.[9] Solving this problem is particularly important for building distributed multimedia systems for the World Wide Web.[70] Atlas is a distributed hyperlink database system that works with traditional servers.[71] Other approaches to the same problem focus on a generic architecture (e.g., through mediators[72]). The software bookshelf uses multiheaded links and an underlying meta-data repository to offer a more flexible, distributed hypermedia system.

In general, the representational frameworks used in knowledge engineering are richer in structure and in supported inferences than those in databases, but those in databases are less demanding on resources and also scale up more gracefully. The bookshelf repository falls between these extremes in representational power and in resource demands. Also, the bookshelf repository is particularly strong in the structuring mechanisms it supports (i.e., generalization, aggregation, classification, and contexts) and in the way these are integrated into a coherent representational framework.

Conclusions

This paper introduces the concept of a software bookshelf to recapture, redocument, and access relevant information about a legacy software system for re-engineering or migration purposes. The novelty of the concept is the technologies that it combines, including an extensible, Web-based architecture, tool integration mechanisms, an expressive information model, a meta-data repository, and state-of-the-art analysis tools. The paper describes these components from the perspectives of three, increasingly project-specific roles involved in directly constructing, populating, and using a software bookshelf: the builder, the librarian, and the patron. Moreover, we outline a prototype implementation and discuss design decisions as well as early experiences. In addition, the paper reports on our experiences from a substantial case study with an existing legacy software system.

The software bookshelf has several major advantages. First, its main user interface is based on an off-the-shelf Web browser, making it familiar, easy-to-use, and readily accessible from any desktop. This aspect provides an attractive and consistent presentation of all information relevant to a software system and facilitates end-user adoption. Second, the bookshelf is a one-stop, structured reference of project-specific software documentation. By incorporating application-specific domain knowledge based on the needs of the migration effort, the librarian adds value to the information generated by the automatic tools. Third, reverse engineering and software analysis tools can be easily connected to the bookshelf using standard Web protocols. Through these tools, the bookshelf provides a collection of diverse redocumentation techniques to extract information that is often lacking or inconsistent for legacy systems. Fourth, the bookshelf environment is based on object-oriented, meta-data repository technology and can scale up to accommodate large legacy systems. Finally, the overall bookshelf implementation is based on platform-independent Web standards that offer potential portability for the bookshelf. Using a client-server architecture, the bookshelf is centralized for straightforward updates yet is highly available to remote patrons.

We consider the software bookshelf useful because it can collect and present in a coherent form different kinds of relevant information about a legacy software system for re-engineering and migration purposes. We also demonstrated that it is a viable technique, because the creation of a large software bookshelf can be completed within a few months by librarians who have access to parsers, converters, and analysis tools. Moreover, the viability of the technique is strengthened in that the bookshelf environment requires little additional software and expertise for its use, thanks to adopting ubiquitous Web technology.

Despite some encouraging results, there are additional research tasks to be completed to finish evaluating the bookshelf technique. First, we are currently validating the generality of the technique by applying it to a second legacy software system. Such a study will also provide a better estimate of the effort required in developing new bookshelves and provide useful insight to bookshelf builders. Second, we wish to study techniques that would allow bookshelf

patrons to extend and update bookshelf contents, as well as adding annotations at public, private, and group levels. This study would ensure that the technology does indeed support the evolution of a bookshelf by its owners and end users. Third, we are working on mechanisms for maintaining consistency of the bookshelf contents and for managing the propagation of changes from one point, for example, a source code file, to all other points that relate to it. Fourth, the bookshelf user interface is sufficiently complex to justify a user experiment to evaluate its usability and effectiveness. Finally, we are currently studying extensions to the functionality of the bookshelf environment so that it supports not only redocumentation and access, but also specific software migration tasks.

Acknowledgments

The research reported in this paper was carried out within the context of a project jointly funded by IBM Canada and the Canadian Consortium for Software Engineering Research (CSER), an industry-directed program of collaborative university research and education, involving leading Canadian technology companies, universities, and government agencies.

This project would not have been possible without the tireless efforts of several postdoctoral Fellows, graduate students, and research associates. Many thanks go to: Gary Farmaner, Igor Jurisica, Iannis Tourlakis, and Vassilios Tzerpos (University of Toronto); Johannes Martin, James McDaniel, Margaret-Anne Storey, and James Uhl (University of Victoria); and Morven Gentleman and Howard Johnson (National Research Council).

We also wish to thank all the members of the development group that we worked with inside the IBM Toronto Laboratory for sharing their technical knowledge and insights on a remarkable software system.

Finally, we gratefully acknowledge the tremendous contributions of energy, diplomacy, and patience by Dr. Jacob Slonim in bringing together the CSER partnership and in launching this project.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Netscape Communications Corporation, or X/Open Co., Ltd.

Cited references and notes

1. H. Lee and M. Harandi, "An Analogy-based Retrieval Mechanism for Software Design Reuse," *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, Chicago, IL, IEEE Computer Society Press (1993), pp. 152–159.
2. J. Ning, *A Knowledge-based Approach to Automatic Program Analysis*, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (1989).
3. G. Arango, I. Baxter, and P. Freeman, "Maintenance and Porting of Software by Design Recovery," *Proceedings of the Conference on Software Maintenance (CSM-85)*, Austin, TX, IEEE Computer Society Press (November 1985), pp. 42–49.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).
5. F. Van der Linden and J. Muller, "Creating Architectures with Building Blocks," *IEEE Software* **12**, No. 6, 51–60 (November 1995).
6. V. Kozaczynski, E. Liongosari, J. Ning, and A. Olafson, "Architecture Specification Support for Component Integration," *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE)*, Toronto, Canada, IEEE Computer Society Press (July 1995), pp. 30–39.
7. J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing Knowledge About Information Systems," *ACM Transactions on Information Systems* **8**, No. 4, 325–362 (October 1990).
8. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
9. L. Seligman and A. Rosenthal, "A Metadata Resource to Promote Data Integration," *IEEE Metadata Conference*, Silver Spring, MD, IEEE Computer Society Press (April 1996).
10. The Apache HTTP Server Project is a collaborative software development effort aimed at creating a commercial-grade source-code implementation of an HTTP Web server. Information about the project can be found at the Internet World Wide Web site http://www.apache.org.
11. G. Valetto and G. Kaiser, "Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments," *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE)*, Toronto, Ontario, IEEE Computer Society Press (July 1995), pp. 40–48.
12. J. A. Zachman, "A Framework for Information Systems Architecture," *IBM Systems Journal* **26**, No. 3, 276–292 (1987).
13. J. F. Sowa and J. A. Zachman, "Extending and Formalizing the Framework for Information Systems Architecture," *IBM Systems Journal* **31**, No. 3, 590–616 (1992).
14. A. Brown and M. Penedo, "An Annotated Bibliography on Software Engineering Environment Integration," *ACM Software Engineering Notes* **17**, No. 3, 47–55 (July 1992).
15. P. Penny, *The Software Landscape: A Visual Formalism for Programming-in-the-Large*, Ph.D. thesis, Department of Computer Science, University of Toronto (1992).
16. H. Müller and K. Klashinsky, "Rigi—A System for Programming-in-the-Large," *Proceedings of the 10th International Conference on Software Engineering (ICSE)*, Raffles City, Singapore, IEEE Computer Society Press (April 1988), pp. 80–86.
17. M.-A. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. Müller, "On Designing an Experiment to Evaluate a Reverse Engineering Tool," *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, Monterey, CA, IEEE Computer Society Press (November 1996), pp. 31–40.

18. M. Chase, D. Harris, S. Roberts, and A. Yeh. "Analysis and Presentation of Recovered Software Architectures," *Proceedings of Working Conference on Reverse Engineering (WCRE)*, Monterey, CA, IEEE Computer Society Press (November 1996), pp. 153–162.
19. J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Co., Reading, MA (1994).
20. S. Tilley, K. Wong, M.-A. Storey, and H. Müller, "Programmable Reverse Engineering," *International Journal of Software Engineering and Knowledge Engineering* **4**, No. 4, 501–520 (December 1994).
21. H. Müller, M. Orgun, S. Tilley, and J. Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification," *Journal of Software Maintenance: Research and Practice* **5**, No. 4, 181–204 (December 1993).
22. L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl*, O'Reilly and Associates Inc., 101 Morris Street, Sebastopol, CA 95472 (1996).
23. T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering* **SE-2**, 308–320 (1976).
24. M. Halstead and H. Maurice, *Elements of Software Science*, Elsevier North-Holland Publishing Co., New York (1977).
25. R. Stallman, "Emacs: The Extensible, Customizable, Self-Documenting Display Editor," *Proceedings of the Symposium on Text Manipulation*, Portland, OR (June 1981), pp. 147–156.
26. S. Mancoridis and R. Holt, "Extending Programming Environments to Support Architectural Design," *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE)*, Toronto, Ontario, IEEE Computer Society Press (July 1995), pp. 110–119.
27. S. Mancoridis, *The Star System*, Ph.D. thesis, Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (1996).
28. D. Kafura and G. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering* **SE-13**, No. 3, 335–343 (March 1987).
29. B. Curtis, S. Sheppard, P. Milliman, M. Vorst, and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," *IEEE Transactions on Software Engineering* **SE-5**, 96–104 (March 1979).
30. E. Buss, R. DeMori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster, and K. Wong, "Investigating Reverse Engineering Technologies for the CAS Program Understanding Project," *IBM Systems Journal* **33**, No. 3, 477–500 (August 1994).
31. D. Schefstrom and G. Van den Broek, *Tool Integration: Environments and Frameworks*, John Wiley & Sons, Inc., New York (1993).
32. *ECMA: Portable Common Tool Environment*, Technical Report ECMA-149, European Computer Manufacturers Association, Geneva, Switzerland (December 1990).
33. S. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software* **7**, No. 3, 57–66 (July 1990).
34. M. R. Cagan, "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," *Hewlett-Packard Journal* **41**, No. 3, 36–47 (June 1990).
35. *The Common Object Request Broker: Architecture and Specification*, Object Management Group, Inc., Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 (December 1991).
36. B. Curtis, M. Kellner, and J. Over, "Process Modeling," *Communications of the ACM* **35**, No. 9, 75–90 (September 1992).
37. "Collaborative Computing," *Communications of the ACM* (December 1991), special issue.
38. *Special Issue on Cooperative Information Systems*, J. Mylopoulos and M. Papazoglou, Editors, IEEE Expert, to appear 1997.
39. G. Kaiser, S. Dossick, W. Jiang, and J. Yang, "An Architecture for WWW-based Hypercode Environments," *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, Boston, MA, IEEE Computer Society Press (May 1997), pp. 3–13.
40. *Visual Reengineering Toolset*, McCabe & Associates, 5501 Twin Knolls Road, Suite 111, Columbia, MD 21045. More information can be found at the Internet World Wide Web site http://www.mccabe.com/visual/reeng.html.
41. R. Arnold, *Software Reengineering*, IEEE Computer Society Press (1993).
42. G. Kotik and L. Markosian, "Automating Software Analysis and Testing Using a Program Transformation System," Reasoning Systems Inc., 3260 Hillview Avenue, Palo Alto, CA 94304 (1989).
43. T. J. Parr, *Language Translation Using PCCTS and C++: A Reference Guide*, Automata Publishing Company, 1072 South De Anza Blvd., Suite A107, San Jose, CA 95129 (1996).
44. P. Devanbu, "GENOA—A Customizable Language- and Front-End Independent Code Analyzer," *Proceedings of the 14th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, IEEE Computer Society Press (May 1992), pp. 307–317.
45. G. Murphy, D. Notkin, and S. Lan, "An Empirical Study of Static Call Graph Extractors," *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, IEEE Computer Society Press (March 1996), pp. 90–100.
46. G. Murphy and D. Notkin, "Lightweight Lexical Source Model Extraction," *ACM Transactions on Software Engineering and Methodology*, 262–292 (April 1996).
47. M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering* **SE-10**, No. 4, 352–357 (July 1984).
48. K. Wong, "Managing Views in a Program Understanding Tool," *Proceedings of CASCON '93*, Toronto, Ontario (October 1993), pp. 244–249.
49. M. Consens, A. Mendelzon, and A. Ryman, "Visualizing and Querying Software Structures," *Proceedings of the 14th International Conference on Software Engineering (ICSE)*, Melbourne, Australia; IEEE Computer Society Press (May 1992), pp. 138–156.
50. L. Wills and C. Rich, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software* **7**, No. 1, 82–89 (January 1990).
51. T. Biggerstaff, B. Mitbander, and D. Webster, "Program Understanding and the Concept Assignment Problem," *Communications of the ACM* **37**, No. 5, 72–83 (May 1994).
52. A. Quilici, "A Memory-based Approach to Recognizing Programming Plans," *Communications of the ACM* **37**, No. 5, 84–93 (May 1994).
53. R. Selby and V. Basili, "Analyzing Error-Prone System Structure," *IEEE Transactions on Software Engineering* **SE-17**, No. 2, 141–152 (February 1991).
54. S. C. Choi and W. Scacchi, "Extracting and Restructuring the Design of Large Systems," *IEEE Software* **7**, No. 1, 66–71 (January 1990).
55. K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching for Clone and Concept Detection," *Journal of Automated Software Engineering* **3**, 77–108 (1996).

56. H. Johnson, "Navigating the Textual Redundancy Web in Legacy Source," *Proceedings of CASCON '96*, Toronto, Ontario (November 1996), pp. 7–16.
57. S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, Toronto, Ontario, IEEE Computer Society Press (July 1995), pp. 86–95.
58. M. Olsem, *Software Reengineering Assessment Handbook*, United States Air Force Software Technology Support Center, 00-ALC/TISEC, 7278 4th Street, Hill Air Force Base, Utah 84056-5205 (1997).
59. P. Devanbu, R. Brachman, P. Selfridge, and B. Ballard, "Lassie: A Knowledge-based Software Information System," *Communications of the ACM* **34**, No. 5, 34–49 (May 1991).
60. P. Devanbu and M. Jones, "The Use of Description Logics in KBSE Systems," to appear in *ACM Transactions on Software Engineering and Methodology*.
61. P. Constantopoulos, M. Jarke, J. Mylopoulos, and Y. Vassiliou, "The Software Information Base: A Server for Reuse," *Very Large Data Bases Journal* **4**, 1–43 (1995).
62. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, M. Nagl, Editor, Lecture Notes in Computer Science 1170, Springer-Verlag, Inc., New York (1996).
63. R. Patil, R. Fikes, P. F. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches, "The DARPA Knowledge Sharing Effort: Progress Report," *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, Boston (1992).
64. T. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition* **5**, No. 2, 199–220 (March 1993).
65. P. Bernstein and U. Dayal, "An Overview of Repository Technology," *International Conference on Very Large Databases*, Santiago, Chile (September 1994).
66. J. Hammer, H. Garcia-Molinas, J. Widom, W. Labio, and Y. Zhuge, "The Stanford Data Warehousing Project," *IEEE Data Engineering Bulletin* (June 1995).
67. H. Jagadish, A. Mendelzon, and T. Milo, "Similarity-based Queries," *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA (May 1995), pp. 36–45.
68. I. Jurisica and J. Glasgow, "Improving Performance of Case-based Classification Using Context-based Relevance," *International Journal of Artificial Intelligence Tools*, special issue of IEEE ITCAI-96 Best Papers **6**, No. 3&4 (1997, in press).
69. "Special Issue: Metadata for Digital Media," W. Klas and A. Sheth, Editors, *ACM SIGMOD Record* **23**, No. 4 (December 1994).
70. Fifth International World Wide Web Conference, Paris, May 1996.
71. J. Pitkow and K. Jones, "Supporting the Web: A Distributed Hyperlink Database System," presented at Fifth International World Wide Web Conference (WWW96), Paris (May 1996).
72. G. Wiederhold, "The Conceptual Technology for Mediation," *International Conference on Cooperative Information Systems*, Vienna (May 1995).

Accepted for publication July 21, 1997.

Patrick J. Finnigan *IBM Software Solutions Division, Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7 (electronic mail: finnigan@vnet.ibm.com).* Mr. Finnigan is a staff member at the IBM Toronto Software Solutions Laboratory, which he joined in 1978. He received the M.Math. degree in computer science from the University of Waterloo in 1994, and is a member of the Professional Engineers of Ontario. He was principal investigator, at the IBM Centre for Advanced Studies of the Consortium for Software Engineering Research (CSER) project, migrating legacy systems to modern architectures, and is also executive director of the Consortium for Software Engineering Research, a business/university/government collaboration to advance software engineering practices and training, sponsored by Industry Canada.

Richard C. Holt *Department of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1 (electronic mail: holt@turing.toronto.edu).* Dr. Holt was a professor at the University of Toronto from 1970 to 1997 and is now a professor at the University of Waterloo. His Ph.D. work on deadlock appears in many books on operating systems. He worked on a number of compilers such as Cornell's PL/C (PL/I) compiler, the SUE compiler (an early machine-oriented language), the SP/k compiler (PL/I subsets for teaching), and the Euclid and Concurrent Euclid compilers. He codeveloped the S/SL parsing method, which is used in a number of software products. He is coinventor of the Turing programming language, which is used in 50 percent of Ontario high schools and universities. He was awarded the CIPS 1988 national award for software innovation, the 1994-5 ITAC national award for software research, and shared the 1995 ITRC award for technology transfer. He is the author of a dozen books on languages and operating systems. His current area of research is in software architectures, concentrating on a method called Software Landscapes used to organize the programs and documents in a software development project. He has served as Director of ConGESE, the cross-Ontario Consortium for Graduate Education in Software Engineering.

Ivan Kalas *Centre for Advanced Studies, IBM Software Solutions Division, Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7 (electronic mail: kalas@torolab.vnet.ibm.com).* Mr. Kalas is a research staff member at the Centre for Advanced Studies, IBM Canada Laboratory. His research interests are in the area of object-oriented design, object-oriented concurrent systems, programming environments, and programming languages. He holds degrees in mathematics and physics, and a master's degree in mathematical physics from the University of Toronto. He joined IBM in May of 1989.

Scott Kerr *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (electronic mail: skerr@cs.toronto.edu).* Mr. Kerr is a research associate and master's student at the Department of Computer Science, University of Toronto. He received his B.Sc. from the University of Toronto in 1996. He is presently working at the Centre for Advanced Studies at the IBM Toronto Laboratory as well as at the University of Toronto in the areas of conceptual modeling and software engineering.

Kostas Kontogiannis *Department of Electrical and Computer Engineering, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1 (electronic mail: kostas@swen.uwaterloo.ca).* Dr. Kontogiannis is an assistant professor at the University of Waterloo, Department of Electrical and Computer Engineering. He received a B.Sc. in mathematics from the University of Patras, Greece, an M.Sc. in computer science and artificial intelligence from Katholieke Universiteit Leuven, Belgium, and a Ph.D. in computer science from McGill University, Canada. His main area of research is software engineering. He is ac-

tively involved in several Canadian Centres of Excellence: the Consortium for Software Engineering Research (CSER), the Information Technology Research Centre (ITRC) of Ontario, and the Institute for Robotics and Intelligent systems (IRIS).

Hausi A. Müller *Department of Computer Science, University of Victoria, P.O. Box 3055, MS-7209, Victoria, B.C., Canada V8W 3P6 (electronic mail: hausi@csr.uvic.ca).* Dr. Müller is an associate professor of computer science at the University of Victoria where he has been since 1986. From 1979 to 1982 he worked as a software engineer for Brown Boveri & Cie in Baden, Switzerland (now called ASEA Brown Boveri). He received his Ph.D. in computer science from Rice University in 1986. In 1992 and 1993 he was on sabbatical leave at the IBM Centre for Advanced Studies in the Toronto Laboratory, working with the program-understanding group. He is a principal investigator of CSER (Consortium for Software Engineering Research), a Canadian Centre of Excellence sponsored by NSERC, NRC, and industry. One of the main objectives of the centre is to investigate software migration technology. His research interests include software engineering, software evolution, software reverse engineering, software architecture, program understanding, software reengineering, and software maintenance. Recently, he has served as program cochair and steering committee member for three international conferences: ICSM-94 (International Conference on Software Maintenance); CASE-95 (International Workshop on Computer-Aided Software Engineering); and IWPC-96 (International Workshop on Program Comprehension). He is on the editorial board of IEEE Transactions on Software Engineering (TSE) and a member of the executive committee of the IEEE Technical Council of Software Engineering (TCSE).

John Mylopoulos *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (electronic mail: jm@ai.toronto.edu).* Dr. Mylopoulos is a professor of computer science at the University of Toronto. His research interests include knowledge representation and conceptual modeling, covering languages, implementation techniques, and applications. Dr. Mylopoulos has worked on the development of requirements and design languages for information systems, the adoption of database implementation techniques for large knowledge bases and the application of knowledge base techniques to software repositories. He is currently leading a number of research projects and is principal investigator of both national and provincial Centres of Excellence for Information Technology. Dr. Mylopoulos received his Ph.D. degree from Princeton University in 1970. His publication list includes more than 130 refereed journal and conference proceedings papers and four edited books. He is the recipient of the first-ever Outstanding Services Award given out by the Canadian AI Society (CSCSI), a corecipient of the most influential paper award of the 1994 International Conference on Software Engineering, a Fellow of the American Association for AI (AAAI), and an elected member of the VLDB Endowment Board. He has served on the editorial board of several international journals, including the ACM *Transactions on Software Engineering and Methodology* (TOSEM), the ACM *Transactions on Information Systems* (TOIS), and the VLDB *Journal and Computational Intelligence*.

Stephen G. Perelgut *IBM Software Solutions Division, Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7 (electronic mail: perelgut@vnet.ibm.com).* Mr. Perelgut received his M.Sc. degree in computer science from the University of Toronto in 1984. His research interests include compiler design and development, software engineering, software reuse, and electronic communications as they affect virtual communities. He is currently a full-time member of the IBM Centre for Advanced Studies and acting as both a principal investigator on the software bookshelf project as well as program manager for CASCON '97.

Martin Stanley *Techne Knowledge Systems Inc., 439 University Avenue, Suite 900, Toronto, Ontario, Canada M5G 1Y8 (electronic mail: mts@cs.toronto.edu).* Mr. Stanley is President and CEO of Techne Knowledge Systems Inc., a startup company formed by a group of researchers from the Universities of Toronto and Waterloo specializing in the development of tools for software re-engineering. Mr. Stanley received his M.S. degree in computer science from the University of Toronto in 1987. His research interests include knowledge representation and conceptual modeling, with particular application to the building of software repositories. He is currently a part-time research associate in the Computer Science Department at the University of Toronto.

Kenny Wong *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., Canada V8W 3P6 (electronic mail: kenw@csr.uvic.ca).* Mr. Wong is a Ph.D. candidate in the Department of Computer Science at the University of Victoria. His research interests include program understanding, user interfaces, and software integration. He is a member of the ACM, USENIX, and the IEEE Computer Society.

Reprint Order No. G321-5659.