IMPLEMENTING AND EVALUATING COMMON REPOSITORY SERVICES

by

BONNIE M. EDWARDS

B.S., Louisiana State University, 1992

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

1995

IMPLEMENTING AND EVALUATING COMMON REPOSITORY SERVICES

by

BONNIE M. EDWARDS

In memory of Suzie and Walter

Acknowledgements

I have been quite lucky in that support for this project has come in many forms, from many people. I would first like to thank my thesis committee of John A. Miller, E. Rodney Canfield, Robert W. Robinson, and Krys J. Kochut for their knowledge, assistance, and support in this effort. I am also very grateful for the support and helpful ideas of my fellow graduate students, especially Sunderraman Krishnan, Ihab Abuelenein, Deepa Ramesh, and Carolyn Giberson.

The support I received outside the academic realm is without question the key to my success and sanity. Most important, I would like to thank my parents and family, without their support and care this would not have been possible. I would like to express my sincere gratitude to all of my friends who have traveled through this with me, especially Leanne Ward, Mandy Connan, and Mindy Hughes, for offering their hospitality, care, guidance, and companionship. Additionally, I would like to offer a special thanks to everyone at Eddie's Attic for their smiles, support, and helpful input.

Contents

Chapter 1

Introduction

Modern computing environments have evolved from simple, centralized systems to highly diverse, integrated systems. Organizations may maintain many database management systems (DBMS), application programs (in-house and vendor supplied), development productivity tools, operating systems, programming languages, compilers, and hardware platforms. Moreover, the client-server architecture has come to the forefront of computing, thus increasing the variety of operating platforms for a system. Considering these intricate and complex information system environments, an organization may find it difficult to account for and identify all of their information resource assets, much less all of the dependencies and relationships between them.

Repository technology attempts to remedy this organizational dilemma. The approach of the repository is simple in nature: the repository acts as a central manager of all of the information resources of an organization. It maintains valuable information about all of the information system assets of an organization and the relationships between them. Therefore, system users can use the repository as a guide to exploring all the resources a computing environment has to offer. Simply maintaining this organizational information is

not sufficient, however. Without distinct services and tools available to work in conjunction with the repository, the true value of the repository to the computing community will remain untapped.

Thus far, little formal research has been done in the area of repository services. The purpose of this thesis is to explore and evaluate the design and implementation issues surrounding repository services. The approach is two-fold. This will be done by first discussing in a theoretical sense basic repository services including overview and design details. In order to explore the issues in an applied manner, a collection of basic repository services was implemented in conjunction with a prototype repository. This development effort serves as a basis for the analysis of the complexity and issues involved in designing and implementing basic repository services. The design and implementation details as well as the results of this effort will be discussed. An overview of repository technology is presented in the opening chapter to put the scope of the project into perspective.

Chapter 2

Repository Background

Modern information system environments are undoubtedly much more diverse and complex than their early predecessors. This is true not only with respect to the sophistication of the software being developed but also to the diversity of tools and platforms being used in the development process. Application development organizations may use many different software design and productivity tools (e.g. Computer-Aided Software Engineering or CASE tools), programming languages, compilers, database management systems (DBMS), operating systems (OS), and machines. Further, they may maintain many in-house developed application programs, individual code modules, and databases. With this breadth of software and hardware components, it is difficult for an organization to account for everything within their information system environment and nearly impossible to track the interdependencies between these components.

The dilemma of accurately managing all of an organization's information system resources, however, just scratches the surface of the problem plaguing modern information systems organizations. Consider the following scenarios.

- A CASE tool oriented development organization uses two software design tools, each from a different vendor, during different phases in the application development life cycle. Since each tool is from a different vendor, the design diagrams created with the use of the first tool are not portable to the second tool. Therefore, design work done using the second tool occurs independently of work done using the first tool. Duplication of effort is inevitable.

- A fast paced application development company supports and maintains several application programs that they market in addition to participating in concurrent development on new projects. There are several distinct departments within the company with a number of developers working in each. The company wishes to track and maintain a complete and up to date inventory and description of all previously developed code so that code reuse can be achieved if at all possible. However, due to the current workload and pressing deadlines, there are no resources available to maintain such a list.

- Software enhancement and maintenance within the database application environment pose a special concern. Database objects and the source code that references and manipulates them are highly dependent upon each other. Consider the case where a database table must be altered in order to add a new field. The first concern of the software maintainer would be to determine which existing application programs are affected by this change and in turn, revise each accordingly. Unless an organization has a tool to automate the tracking of dependency relationships between application programs and database objects, the task of analyzing the impact of such a change may be a painstakingly long process.

- For a single application program development project, each of the five components of the application are assigned to separate development teams. Two of these components will be shared with existing application programs. A simple source code control and versioning system is in place that allows individual source code files to be managed. However, this mechanism has proven itself insufficient in a highly integrated, reuse oriented development environment. Since it is crucial to relate the correct version of each individual component with a specific version of the application program, the developers wish to have change control enacted for the entire application program which supports the tracking of this dependency information. Further, they wish to have an automated and efficient means of performing builds of the executable file for the application program.

It is clear from these examples that many important data management issues in the application development environment must be addressed. Solutions have been attempted to remedy all of these problems. The key problem with these approaches, however, lies in the scope of their solution: their focus has been limited to information management within a single software component type or a collection of tightly related component types [18]. Common approaches include data dictionaries and encyclopedias. *Data dictionary systems* maintain descriptions of data elements or groups of data elements for a given database, project, or application program. Traditionally used in the implementation and maintenance phases of the software life cycle [6], data dictionary systems support either a single product or group of vendor specific products. Further, they usually have proprietary, closed interfaces [18]. The advent of CASE tool technology brought upon an advancement to the data dictionary system in the form of the *encyclopedia*. Encyclopedias provide capabilities for managing graphical model depictions. Further, they form the basis for

the integration of CASE tools. Although this integration of CASE tools takes a step in the direction to a more broad approach to information management, this integration is typically only possible across a set of vendor specific tools [21].

Repository technology attempts a global solution to the information management problem. Its primary purpose is to organize all of the information within a company's information system environment and relate it to its various software components (such as source code, applications, and databases) in such a way that it is accessible to all users who need it regardless of the means by which they obtain it [21]. Even though the beginnings of repository technology development are rooted in the integration of CASE tools supplied perhaps by different vendors, its importance as an overall information resource management system cannot be understated.

The three primary aims of repository technology are to provide a medium for the integration of CASE tools, to promote a highly integrated software engineering environment, and to provide users with a library of all information system components. There are numerous other benefits that a repository can offer, however, including the following [13]:

- Supporting multi-user integrated software development environment.
- Enhancing overall information sharing.
- Eliminating redundant corporate data.
- Increasing the reuse of existing software.
- Simplifying software maintenance.
- Simplifying information migration and conversion.
- Increasing overall development productivity.
- Providing impact analysis capabilities.
- Providing version management for key software components.

To better understand the true potential of the repository, it is essential to understand its underlying structure. At the most basic level, the repository is composed of a data storage area and a management system which allows for the manipulation of the data. It is intended to be a logically centralized library of an organization's information assets and the relationships between them, although its implementation may be either physically distributed or centralized. Typically represented within the data storage area could be the following varieties of data [13]:

- Logical data and process models
- Physical data definitions
- Source code and documentation
- Business data and business rules
- Project management data
- System auditing data
- Life cycle model
- Security information
- Inventory of existing development resources
- Interrelationships between all above items.

For most of these varieties of data, the data itself is fully stored within the repository. In other cases, however, it may not be. What is generally stored within the repository is descriptive data about information resource assets. This abstracted view of data is referred

to as *metadata*. To better illustrate this concept, consider how a repository identifies a file containing a particular version of documentation for an application program. Instead of including the file itself within the repository, information that adequately describes the contents of the file and its location is stored. For instance, the name of the host machine, directory path, and file name that contain the file are stored as well as the owner of the file, the creation date, the date the file was last modified, the version number of the file, and a textual description of the contents of the file. In addition, relationship information linking the document file to the specific application program or application program version is also recorded. Using this information, a user of the repository could gather enough information about the file to quickly access it, if required.

In the process of designing the repository data storage structure, each facet of the information system realm (e.g., database systems and application life cycle) is first modeled using a common modeling technique (such as Entity Relationship (ER) or Object-Oriented Modeling) [13]. Then, all of these submodels are combined to form a global information system model which represents all of these subcomponents as well as the relationships between them. At the heart of the repository, this global model is commonly referred to as the *meta model*. The goal in designing the meta model is to represent all facets of an information system in the most general sense.

Notice how the data store can be viewed at varying levels of abstraction (Figure 2.1). At the base level is the raw information system data itself (such as files and applications), which is not directly stored in the repository. This raw information is then abstracted into descriptions at the metadata level. Data at this level is directly stored in the repository. Above the metadata level is the *meta model* level. The meta model level defines the data constructs that will store the metadata; that is, it defines the *schema* of the metadata.

Figure 2.1: Four-level Repository Architecture.

At the topmost level, the schema for the meta model is defined. This uppermost level would consist of constructs which represent traditional database objects such as entities, attributes, and relationships.

Although the meta model is the most essential part of the repository, it is by no means the only component. Tannenbaum points out the primary components of a generic repository as the following [21]:

1. **Repository Meta Model.** Encompasses descriptions of all information tracked and accessible using the repository.

2. **Underlying DBMS.** Provides an organized and manageable means of storing the repository contents.

3. **Repository-Supplied Utilities.** Provides repository specific functions. These could entail administrative functions (repository loading functions) or development process or system based services (configuration management, version control, impact analysis, and usage analysis).

4. **Interface.** Provides multiple routes of accessing the contents of the repository. Included in this could be access views or templates, tool interfaces, and application program interfaces (APIs).

5. **Repository Security.** Provides some level of control over the contents of the repository. This could be simple DBMS supplied security control or custom tailored (or configurable) security control.

Since a primary aim of the repository is vendor independent tool integration, it is essential that standards exist to define how tools and other applications will communicate with the repository. Several standards have been developed in the recent past that describe both the structure and functionality of and import/export specifications for repositories. Included in these proposed repository standards are ATIS (A Tools Integration Standard) [21], PCTE (Portable Common Tool Environment) [25], and CDIF (CASE Data Interchange Format) [4]. Much of the early repository standards development can be attributed to IRDS (Information Resource Dictionary Specification) [1]. Unfortunately, no single standard is unanimously accepted. Thus, a key problem remains in that different repository products will adhere to different standards. As a result, complete information system integration will not always be possible.

Despite the plethora of standards existing and the general lack of agreement upon a single definition of a repository, several basic characteristics of a repository are generally agreed upon [21, 22].

1. *A repository is an integrated holding area.* The repository's meta model should be designed such that information systems and all related components are correctly and completely representable as well as the the relationships between the components.

2. *The input, access, and structure of a repository and its contents should be vendor independent.* A repository should have an open architecture and supply a standardized interface to allow for interaction with tools and products from a variety of vendors.

3. *The repository's meta model should be extensible.* To enable repository customization, the meta model should be extendible. That is, if additional constructs are necessary to maintain the metadata for a particular tool or application, there should be some mechanism to allow these constructs to be added to the meta model. This flexibility is provided via the meta-meta model level.

4. *The contents of a repository should be retrievable via predefined views or templates.* Access to the metadata must be flexible and tailorable. These access paths must be both definable and maintainable.

5. *The contents of the repository must be versionable and subject to user-implemented security.* The repository should supply version management for key software items such as databases, applications, design models, and files. Further, the data within repositories should be protected via some security mechanism, the scope and method of which should be tailorable.

6. *The repository product should include the capabilities required to perform all of the preceding tasks on a highly functional basis.*

Although repository technology is still rather immature, there are several repository products on the software market currently. Two notable contributions are Digital Equipment Corporation's (DEC) distributed repository product, CDD/Repository [7] and IBM's centralized Repository Manager/MVS [9, 14]. Several other products are currently available

as well [15].

Whether a repository is custom built or purchased, phasing it into the information system environment of an organization is no trivial task: it is quite an investment in terms of money, time, and effort. Several key issues must be dealt with in the implementation process. Among the most notable of these are the following [15]:

- Repository population techniques,
- Repository administration,
- Repository services,
- Repository security,
- Repository interfaces.

The remainder of this thesis will focus on the issue of the evaluation, design, and implementation of services for the repository.

Chapter 3

Repository Services

The repository provides an ideal framework for managing high level details of all of an organization's information assets. Providing this structure, however, is not quite enough. Certain services, or repository functions, are necessary for managing, manipulating, and analyzing the contents of the repository. Without such services provided, the repository simply functions as a glorified data dictionary. In this stagnant state, the full potential of the repository is left untapped. These services are at the heart of its applicability to users, developers, administrators, and managers.

What services should accompany a repository? This problem is best approached by first determining what services a repository can potentially provide then discriminating amongst these to determine which would be most valuable for a particular organization. A close analysis of the basic reasons for implementing a repository will shed some light on this question.

In the most basic sense, the repository acts as a manager. Managers must cater not only to those which they manage (employees in the business sense) but also to those reaping benefits or services from those which are managed (customers in the business sense). This

appeasement is only a minor role for a manager, however. In management, there are several key productivity goals:

- To attain maximal resource use,
- To achieve tasks in minimal time,
- To achieve tasks at a minimal cost,
- To maximize output and profits.

The first step towards achieving any of these goals is to clearly identify all resources which are managed. Organization is key to this task. A manager must know what resources are available for use as well as their capabilities, limitations, interdependencies, and workload. From this, over allocated and under allocated resources can be identified as well as potential productivity bottlenecks. Further, with an omniscient view of all resources and tasks, a manager can identify and eliminate any redundant tasks, and in turn maximize the reuse of work and information. Therefore, having the ability to track all resources and tasks as well as analyze the relationships between them is key to attaining maximal productivity in the business world.

This scenario can be easily extended into the computing realm with the repository as the acting manager. Similar to the business world, the repository must ensure support for both the resources that are managed and those who benefit from their management. The repository proper is primarily concerned with the resources that are to be managed. More specifically, these resources are information resources which could include source code, databases, productivity tools, application programs, and compilers. The repository meta model together with its underlying DBMS provide support for the storage and maintenance of all information resource metadata.

On the flip side, the repository must support the needs of those who can benefit from the management of the information resources. The repository proper does not inherently

support the needs of these beneficiaries other than by providing them access to the contents of the repository. Although simple access to the contents is crucial to these users, it alone cannot ensure the productive use of the resource metadata. Herein lies the need for specialized repository services. These services are focused on the needs of software developers, end-users, system administrators, and corporate managers. Each of these user groups have distinct and important needs that a repository can and should satisfy.

Software developers primarily revise and create new software objects (such as source code and databases). Therefore, they will benefit from services that ease the management of the software life cycle. This could include code inventory, change control, version management, configuration management, impact analysis, release management, and life cycle tracking services.

System administrators will benefit most from performance analysis and auditing services. Information gathered from these services could allow the administrator to detect and correct points of low performance within their computing systems. The administrator will also be interested in services that aid in securing user access to the various information resources in the system.

End-users will be most interested in basic access to the repository contents. For instance, they may wish to locate an application program best suited for a certain task. Upon finding a viable application, they will want to know how to execute the program and perhaps where to find additional user documentation for it.

Corporate managers have more diverse needs than the other user groups. They will benefit from project management, resource allocation management, software inventory reporting, and project progress tracking services. Further, summary information from which high level strategic decisions can be based would be invaluable for a manager.

Though there is a breadth of potential repository services, not every organization will need this full collection. Not all computing organizations have the same mission or scope. Some are strictly development based; some are not. Some deal with database application development. Some emphasize CASE tool based design. Some employ object oriented design paradigms while others focus on more traditional approaches. Further, an organization may already have tools that perform many of the services described above and do not intend to replace them. All of these variables will in turn determine what services an organization will want to implement and at what level.

Repository services can be grouped into two general categories: software development and maintenance services and system administration and maintenance services. Software development and maintenance services provide for management of key development and maintenance related information resources tracked by the repository. Services included in this group are version management, configuration management, change request, project management, impact analysis, type management, modification tracking, and software inventory reporting services. System administration and maintenance services focus on gathering information for productivity analysis and system auditing. Services included in this group are usage analysis, user authorization, and user subscription services. Each of the following sections will discuss and outline a single service, in turn providing overview information, feasibility determination direction, and design issues.

3.1 Version Management

The primary aim of the repository is to manage and allow maximal sharing of information resources across many interfaced tools and users. Therefore, revisions of this managed

information can occur along many paths, by any number of tools. It is essential to track the revision history of these managed objects. The predominate means of accomplishing this is through version management. Version management allows a specific software item to have many instances each linked to one or more of the other instances in a specific and clearly defined sequence. There are two primary benefits to versioning specific software items. First and most important is the maintaining of historical data. The maintaining of a revision history has two immediate benefits. First, code reuse can occur in that specialized instances of a software item can exist in conjunction with each other. Further, the evolution and revision of software items can be traced throughout their life cycle. Support for parallel revision is also offered by version management systems.

Version management can occur at two levels within the repository. Many interfacing tools such as source code control and DBMS systems have internal version management for the data they maintain. In this case, the repository would simply need to manage the information about these versions but not enforce a specific versioning scheme; it will be defined and controlled by the externally interfaced tool itself. The primary concern when implementing version management on this global level is how to represent and maintain the versioning schemes used by different systems. This will be best accomplished by supporting a generic versioning scheme that supports a flexible version identifier assignment and a relationship that defines the successor-predecessor ordering of the versions. This could be extended to support storing textual descriptions of the changes made between two consecutive versions.

This simple version management framework is not always sufficient, however. Many interfacing tools do not support an internal versioning for their managed data [21]. Therefore, it may be necessary for the repository to implement a more strict version management (or

version control) mechanism. When the repository acts as an integrator of data across tools, the data itself will be stored in its entirety within the repository. For example, consider a data model that can be retrieved and updated by two separate development productivity tools. Two users should not be allowed to alter the model at the same time, regardless of what tool is being used to access the model. Further, a history of changes made to the model as well as information identifying the tool and user should be maintained. Version control will need to be enacted at the repository level to manage this change process and enforce these two requirements.

The following sections will outline the issues and facets of version management at the repository level.

3.1.1 Basics of Version Management

A version of an item can be viewed as a physical state of the item at a given frozen point in time. More specifically, a version is an instance of an software item. By implementing a given item as versionable, we allow for a history of changes to the item to be maintained. Thus, one can trace changes throughout the life cycle of the item. For example, a specific bug fix in a code module can be traced to being implemented within a specific version of the item. Or, perhaps, we could compare the original version to the latest version to view all changes enacted throughout the lifetime of the item. In addition, if a given change is made in one version of a code module then it is later deemed useless, an earlier version can be recovered so as to bypass the unwanted change. There are numerous other advantages to the versioning of software items as well.

Version management encompasses more than just allowing for an item to be versionable. According to [26, 19], version management should answer questions such as the following:

- How should the system be structured so that different systems can be built to meet the requirements of different users?
- How should an old version of an item or system be preserved?
- How can a version of the system be built so that it contains certain fixes but not others?
- How may two developers work on the same item at the same time?
- How can many versions of an item be stored efficiently?

Insights into these issues will be presented in subsequent sections.

3.1.2 Versionable Items

Any software item under complete control of the repository should be explicitly versioned. This includes data that is intended to be shared by tools interfaced to the repository with the repository as the sharing mediator. In a CASE-based development environment, design models would likely fall into this environment.

It is important to make these items versionable considering the high level of integration and data sharing in a repository environment. Many different tools interfaced to the repository system will have read and/or write access to common data collections. With so many potential routes and modes of revision, it is essential to track all version instances as well as the details associated with each revision.

Further, if an organization wishes to use the repository as a central controller of the application development process, all potential application components should be versionable. This could include source code modules, definitional modules, object code, documentation, design plans, test data, databases and database objects. In any case, if the item under

version control is itself a composite item, all potential components of the composite item must be versionable as well.

There are two primary issues to consider when deciding at which level to implement versioning. First, the meta model of the repository must be conducive to supporting versioning of all key software item types. To minimize meta model complexity, however, only key high level items such as modules, documents, and data models could be made versionable.

Second the space required to store and maintain all versions of an item must be considered. If all possible software items were made versionable, the repository could explode in size and may not be easily managed. For a simple non-CASE oriented software development environment, it may suffice to simply make versionable the code and definitional modules, documentation, and applications. Decisions regarding which items to make versionable will highly depend upon the nature, needs, and complexity of an organization's software development environment as well as the meta model of the repository itself.

3.1.3 Identifying Versions

A version control system must provide a specific strategy for identifying (or enumerating) item versions. The simpliest approach is to assign a single version number (e.g. 1, 2, 3...) to each instance of an item [26] (see Figure 3.1). This is not very flexible or practical. There is no inherent meaning between two successive version numbers and there is no capability for more than one line of item development.

This single number approach is commonly replaced with a multi-level version identifier. One common approach is to split the version identifier into two parts: a release number and a revision number [26]. In this approach, increments in the revision number indicate minor changes to the item which occur more frequently whereas increments in the release

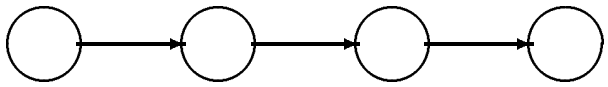

Figure 3.1: Single Component Numeric Versioning Scheme

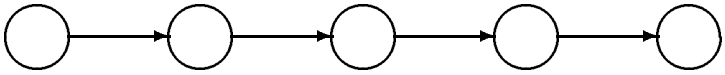

Figure 3.2: Two Component Numeric Versioning Scheme

number indicate major changes to the item (or perhaps that the item reached a sufficiently high level of approval to be released to the public). Refer to Figure 3.2 for an illustration of this versioning scheme.

This still may not be sufficient, however. This linear approach neglects to support two valuable situations. It does not support concurrent development on the same instance of the same item. Further, it does not support the existence of different instances of an item specialized for certain applications. To enable this, a third component of the version identifier called a *variant* can be used [26]. Therefore, the version identifier will have three components: a release number, a revision number, and a variant number. Refer to Figure 3.3 for an illustration of this versioning scheme.

Temporary variants are commonly used to branch off of the main progression line of revisions. If the branch is used to facilitate the concurrent revision of an item, the item will be eventually merged back into the main version line. Permanent variants, however, can be used for different but equal item or system requirement specifications. For example, if an executable version of an application is needed to execute upon both a UNIX and VMS

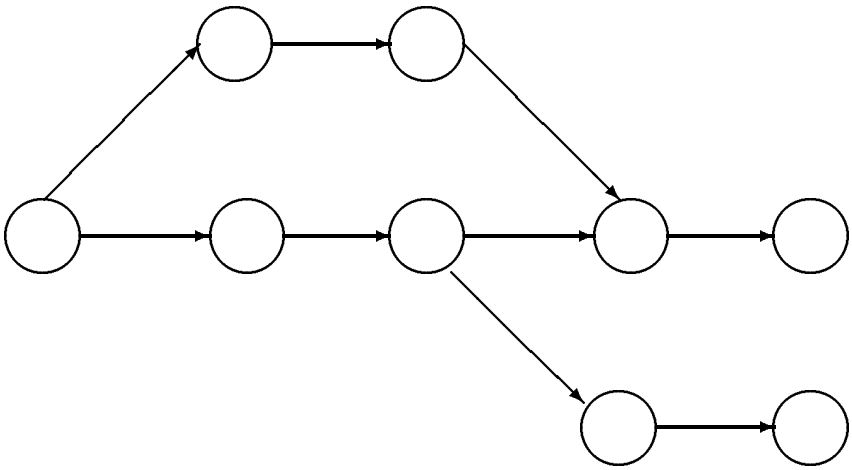

Figure 3.3: Three Component Numeric Versioning Scheme

platform, a permanent variant will exit for each of these specifically targeted application versions.

The rigid three level numbering scheme cannot support multiple levels of branching. That is, a single branch can exist off the main version line but a branch cannot exist off of another branch. Versioning schemes that support multi-level branching involve version numbers with a dynamic number of components. There are two common approaches that support multi-level branching.

The first approach is purely numeric [26]. It uses a single component version number along the main versioning line. For each branch extending from the main line, an additional version number component is appended. If a branch originates from an existing branch, another version number component is appended. Refer to Figure 3.4 for an illustration of this versioning scheme. This approach has one limitation, however. Only one branch can originate from any given item version.

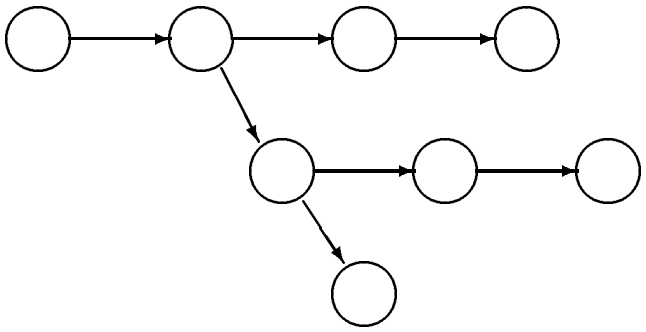

Figure 3.4: Dynamic Component Numeric Versioning Scheme

The second dynamic approach is the most flexible and it does not exhibit the limitation of the purely numeric dynamic version scheme. Versions are identified by a combination of both numeric and textual components. A progression of version along the main line or any single branch are identified by a sequential ordering of single component numeric version numbers. Each branch is then identified by a unique alphanumeric string. Refer to Figure 3.5 for an illustration of this versioning scheme. Note that a particular version will be identified by an absolute path specification from the top of the version tree, for example, `/main/fix_a/2`. This versioning scheme is implemented within Atria's ClearCase configuration management system [2].

In deciding which versioning scheme to implement, several issues should be considered. If an organization is relatively small and the primary reason for versioning is for revision history, one of the simplier versioning schemes may suffice. However, if versioning is intented to facilitate parallel revision or functionally distinct variants, one of the more sophisticated versioning schemes should be implemented. Complexity of the design of the versioning scheme and the space and overhead required to maintain the version history are also important concerns.

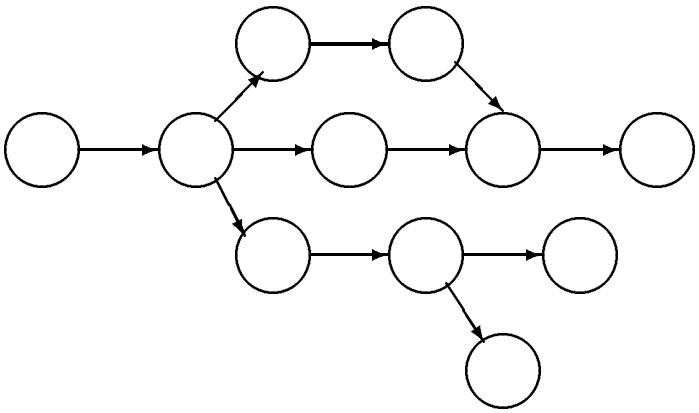

Figure 3.5: Dynamic Component Alphanumeric Versioning Scheme

3.1.4 Tracking Change For Simple Items

Associated with any versionable item should be an indicator of its level of acceptance or testing [3, 26]. Typical approval status indicators could include *working*, *unit tested*, *system tested*, and *approved* [26]. This metadata will be of great use to inform all developers what state the item is in. Lack of this knowledge can result in many obstacles for both the developers and maintainers. When an item is in the *working* status, it is expected that changes will occur quite frequently. Each of these small changes may not require a new version of the item, however. At this point, the item is usually only visible to one developer and minor changes are not crucial to retain. However, when the status is at an approval level higher than *working* and the visibility of the item is in turn increased and any change to the item should render a new version.

To preserve a history of changes to an item, all but the most recent version in a history of changes (usually called the *current* version) should be *frozen* from further development. By *frozen*, it is meant that the item instance can undergo no additional change. This can be implemented at the operating system level by setting the file access permissions of version

to read-only. If a versioning scheme that allows branching is used, it is possible to create a branch originating from a frozen version, and in turn, perform revisions upon this branch. These are typically temporary variants that will eventually be merged back into the main line of descent of the version history of an item.

Since each change to a highly visible item instance results in a new version of the item, a sequence of related revisions is created. This sequence is typically called the *version history* of the item. To illustrate the version history graph more clearly, consider the purely numeric three component versioning scheme based upon a release, revision, and variant number being assigned to each instance of an item. In considering the historical line where only the revision number changes as you progress down it, we are viewing the *main line of descent* in the version history of an item. On the other hand, when progressing down a variant branch, only the variant number is incremented. The user should be able to view the version history of a particular item. Therefore, the version management system should provide either a graphical or textual represtation of the version history tree for all managed items. Refer to Figure 3.3 for an illustration of a version history diagram.

3.1.5 Revising Items

The most common model employed by version management systems for controlling the revision of managed items is the check-in/check-out model. In this scenario, all items controlled by the tool are held in a reserved area called a *library*. Two controlling operations exist, namely, the check-in operation and the check-out operation.

When an item needs to be revised, the check-out operation is issued. In a copy in/copy out based version management system, this essentially copies (or makes visible) the current version of the item into the developer's workspace. Upon a successful check-out, the

developer is free to make any changes to the item: it is out of the reach of the version management system. Once all changes to the item are complete, the developer will issue the check-in operation. Typically, checking the item into the library will result in a new version of the item. Upon checking in an item, the version management system should prompt the user for details regarding the changes that were made. This allows important historical metadata to be maintained about each instance of the item. Change history reports that display this historical metadata for all versions of an item should be available for users. Often, a developer may check-out an item, make several changes to the file, then deem the changes made to be unwanted. An un-check-out feature should be available to disregard any changes made to the code while it was checked-out.

It may be beneficial for the version management system to have the capability to send notification to all interested developers at important points in the life cycle of an item. Developers may be interested when particular items are checked in or perhaps when the acceptance level of an item has been changed. A repository based version management system can allow for this service since detailed information about all system users, including their electronic mail addresses, are generally maintained.

For items under version control of the repository that will not be ported between tools, the repository should provide a user level interface (menu, graphical, or command line based) for the revision of these items. However, for items that are intended to be shared by interfaced tools, a low level interface (API) should be supplied that enables repository information retrieval and storage through the interfaced tools.

3.1.6 Concurrent Revisions

There are two general approaches to the problem of two developers wishing to modify the same version of an item in parallel. A strict and limiting but most easily implemented choice is to use a file locking technique. If a developer checks out a given version of a file in update mode, the file is then locked so that no other developers can make changes to the file until it is checked back in. The second approach is more broad and open but more complicated to implement. In this approach, concurrent revision is allowed to occur in a regulated fashion via a temporary branch in the version history of an item [26]. Eventually, the branch will need to be merged back into the main line of version descent.

To aid in this process, the version control system should provide a merging facility. The traditional text merging algorithm is as follows [7]. A three way comparison between the two versions to be merged and their closest common ancestor is performed. These three files are scanned and compared line by line in a parallel fashion. If at a given point, all three files are the same, that text is included in the merged file. If both versions are different at one point, but one is the same as the common ancestor, the text from the version which is different is included in the merged file. However, if all three files differ at a particular point, a *collision* occurs. The merging facility cannot determine without manual human intervention which of the versions should be included in the final merged file. If no conflicts occur, all changes enacted in each version will be included in the merged file. Traditional merge tool support text file merging exclusively. More advanced, application specific merging tools support a broader base of file types including binary files.

One of the main limitations of merging algorithms is that they merge text files blindly and therefore cannot understand the meaning or semantics of the files being merged. As a

result, problems can frequently occur [7]. Although useful to some extent, merge tools are definitely no panacea. All results from automated merges, even conflict free ones, should be checked for correctness.

3.1.7 Version Storage

Due to the highly transient nature of software development, a multitude of versions will be created in the lifetime of a software item. Considering all of the software development activities of an organization, it will be nearly impossible to store every separate instance of the item in its entirety. Disk space is too much at a premium for this. To remedy this, most version management systems provide a minimal storage technique for the maintaining of all instances of an item that are exclusively managed by the system.

The most common solution to this problem is to store each individual version as a *delta* or change from a previous version. One of the first and most widely recognized packages implementing this delta storage technique is the Source Code Control System (SCCS) [17]. SCCS's approach is quite simple in nature. The first version of an item is stored in its entirety. Each subsequent version, however, contains only the textual differences between it and its immediate predecessor. SCCS bases differences on a line by line basis rather than upon a character by character basis. Any time that a version other than the initial version is requested, all deltas between the first version and the requested version are applied to recreate it. This is referred to as recreating using *forward deltas*. As can be expected, time constraints in recreating item versions distinct from the initial version can definitely be a problem with this approach.

Another popular implementation is in the Revision Control System (RCS) [23]. RCS is quite similar to SCCS in its delta storing but it uses *reverse deltas* to recreate versions.

That is, the most current version is stored in its entirety and any previous versions must have deltas applied to recreate them. There is great time savings in this approach since most often it is a terminal version of an item that is desired.

One of the main problems with past approaches to delta storage is that the functionality is only provided for text files and not for binary or data files. This is changing to meet the more complex needs of software developers [26]. Delta storage is not without its drawbacks: there is a definite tradeoff between recreation time and disk storage space. For more simple development needs (i.e. fewer revisions) time constraints may not be a problem but for very complex development projects, this approach may not be desirable. As an enhancement, delta storage can be accompanied by a data compression technique to further reduce the space needed [2, 26].

3.1.8 Change Control

In contrast to version management which manages the items which result from change, change control manages the overall process of changing and maintaining software items [26]. Although this is a very useful component of any development based organization, it is not a crucial repository service to provide. Aspects to consider in change control are the underlying formal procedures for specifying and implementing change, the format and content of the change request, the methods of transferring a request through its life cycle, and user notification at key points in the life cycle of the change request.

Change Requests

In any active application development environment, the revision of code is frequent and commonplace. It is best not to carry out these revisions haphazardly. Changes are typically

initiated through formal requests which provide a means of tracking the entire life cycle of change, bug-fix, and enhancement requests. Although there are many stand-alone products that offer this functionality, the repository can offer a more integrated and broad change management and tracking system. Though not a crucial repository service, it definitely would make the life of developers and especially maintainers much easier.

Change procedures are highly organization dependent. Therefore, a change request system must strive to employ a non-rigid methodology; as a service, its aim is not to control but to aid. Many organizations will have their own specific system for proposing, reviewing, and enacting changes to software items already in place. At best, the change request system should support a general change control protocol but not require it in a strict sense. Further, its use must be wholly optional.

A change request service implemented in conjunction with a repository should support change request for key software items such as code modules, documents, and database objects. The approach is simple in nature: provide an automated tracking system for change request forms. A good implementation would be to allow for the automated management of general change requests much like standard paper-based control system forms for change [26]. Any time a change to an item is desired, a user will issue a change request for the item. Within this change request will be information such as the reason for the change, the description of the change needed, the originator of the request, the date of the request, as well as other necessary details. The change request is then moved from a proposed status to be under review, possibly by some review board. In this state, a reviewer will assess the cost and potential impact of the change upon existing items and systems then either approve or reject it. Information regarding the approval or rejection and costs would be noted in the change request at this point. In the final step, if the request is approved, the

change is implemented. Once again any details concerning the actual implementation of the change is recorded in the change request. Thus, the change request system serves as a manager of formal changes to key software items managed by the repository. As illustrated above, the change request has a life cycle just as any other software item and usually exists in a state of *draft*, *under review*, *approved* or *rejected*, or *implemented* [26]. This is a good model for organizations desiring a more controlled software maintenance environment.

The change request system should also provide reporting capabilities. A user should be able to generate reports on the details of a single change request, a listing of all pending change requests, and a listing of all change requests associated with a particular software item. Other specialized reports can supplement this basic selection.

3.2 Configuration Management

In a highly integrated development environment, the management of items only on a base atomic level is not sufficient. Version management provides a framework for managing individual software items such as code modules and documentation files, but provides no inherent management support for collections of these atomic items. Configuration management is an extension of version management that manages the change processes not only with respect to atomic software items but also composite objects or *configurations*.

The individual components of a configuration may come from many separate sources—design models from a CASE tool, source code managed by an outside source code control system, and tables within databases managed by a specific DBMS. That is, the components may be a mixture of non-versioned items, items versioned within an external system, and items explicitly versioned within the repository. Therefore, the representation of the com-

ponent versioning scheme whether it be internal or external should be flexible and unified. An important point is that since the structure of a configuration is completely controlled by the repository, the repository must provide version control for the configuration itself—this will not be done externally.

The repository provides a centralized store to integrate all of the data from these component contributors. Therefore, it is desirable that functionality is provided to manage the relationships between all components in a project configuration. This is an essential repository service that exposes the true integrated nature of the repository.

The following sections will outline the many issues and facets of implementing a configuration management system both in general and with respect to a repository based environment. Three key services fall under the umbrella of configuration management, namely, automated configuration builds, release management, and project management.

3.2.1 Configuration Management Basics

To accurately define configuration management, it is appropriate to first define what a configuration is. In the most basic sense, a configuration is a collection of elements that fulfill some particular purpose. It is a composite item which is itself a collection of elements. Consider an application program. A particular instance of the application program is made up of a set of source elements each of a particular version. Also, within the application configuration would be all of the corresponding object files, executable files, user and system documentation, test data, data and process models, design specifications and plans, and project plans. A relational database configuration would be composed of a collection of tables, indexes, rules, triggers, views, and defaults as well as design models and documentation. Dependencies between these components is the fundamental relationship that

configuration management systems must control.

Version management should not only apply to simple or atomic items but also to composite items or configurations. The version management concerns for composite items are somewhat different and more complex than those for simple items. Three major issues are at the forefront when considering the versioning of configurations:

- How to manage the components of a configuration version?
- When to create a new version of a configuration?
- Whether to bind the constituent items to the containing configuration statically or dynamically?

A strategy on how to define which version of an item belongs to a given composite item version must be clearly defined. A reasonable approach is to allow for two types of configuration bind dependencies: static or dynamic [26]. If a version of a simple item is statically configured to a specific composite item version, the dependency link is immutable unless manually changed. The overhead of maintaining static configuration linkages is negligible. On the other hand, the configuration dependency link of a simple item which is dynamically configured to a composite item version is determined at any time as the current, terminal version of the simple item on its main line or on a particular branch of version descent. This approach provides quite a bit of flexibility for software developers but imposes a processing overhead to maintain the linkages. Anytime that an item dynamically configured to a configuration has a new version created, whether it is subject to internal or external version control, the configuration structure must be updated to include the new version. This may produce a further side effect of a new version of the configuration as well.

A simple policy on when to create a new version of any simple item has been presented. It is not quite as simple for composite item versions, however. The main difficulty lies in that

its constituent items can be constantly changing. If all of the components of a composite item are statically bound to it, the problem of a new version is simple: a new version is likely be created when one of the static links is changed. However, if any constituent items are dynamically bound to the composite item a question arises. Must we create a new version of the composite item each time one of the dynamically bound constituent items is revised? Not necessarily. Considering disk space and processing overhead, it may not be reasonable to constantly create new composite item versions in these cases.

There are three possible approaches to this problem. One, create a new version of the composite item each time a dynamically configured component item is revised. As stated before, this is a very time and space consuming approach which is likely to be infeasible for large projects. Two, create a new version of a composite item only when one of its dynamically configured constituent items has had an upgrade in approval status. This approach seems quite feasible and not too difficult on time and space constraints. Three, create a new version of a composite item only when the status of the composite item has had an upgrade in approval status. We can define the approval status of a composite item to be at most the lowest status attained by all of its constituent item versions. This approach is quite limiting and will not result in many historical frozen configuration versions of the composite item. Considering these options, the second one seems to be the best solution. Once again, the needs and complexity of an organization's development environment will determine what is best.

Check-in/check-out should apply not only to simple items but also to composite items. For example, if a member of the maintenance team needs to check out an earlier version of the configuration for testing and troubleshooting, he or she should be able to check out all items in the configuration at one time. In the case of an application configuration, this

would include all source code modules, definitional modules, and possibly documentation files.

3.2.2 Building an Application Configuration

It is not feasible for a developer to have complete knowledge of all files and module versions that compose an application configuration at their fingertips. This problem is further compounded when development occurs in a large, fast paced software development environment. Considering that many organizations which lack automated configuration management support have employees devoted solely to the task of configuration management and builds, the importance of an automated build facility is clear. Configuration management systems can help remedy this problem by providing an abstracted view of building applications to the developer in that only an application name and a specific version number are required information in order to build an application.

The basic elements that are needed in the building of a configuration are the related source items, the system model, version selection of the related items, and the derived or object items [10, 26]. This information is precisely what the repository in conjunction with a configuration management service strives to manage and maintain. Therefore, all of the dependency information is easily obtained. Not only are the dependency links stored within the repository but also information about those dependencies. For the purposes of builds, important supplemental information provided would include linker and compiler information as well as the corresponding command line options. The X-windowing system provides *makedepend*, a utility for determining source file to header file dependency information for use with a *makefile*. This utility is limited in scope, however, since it only determines source file to include file dependencies given that the user correctly identifies the source

items. Therefore, prior to using this utility, the user must accurately determine all source and object files that a build depends upon as well as their correct versions. The utility then scans each source file for **#***include* directives from which to build the source file to include file dependencies. The laborious and potentially error-prone task of identifying the correct versions of all object and source file components to be included in an application build is precisely what automated configuration build facilities strive to avoid.

Another problem solved by an automated approach is ensuring that the appropriate version of all component source items is included in a configuration. This is facilitated through the support for both static and dynamic configuration links between configurations and their corresponding source elements.

As far as how to accomplish the automated builds, there are a few options. The main idea behind building an application configuration is to perform individual compilations on all of the subsidiary source files then link the related object files and object libraries together, either statically or dynamically as determined by the command line parameters, to form the final executable file. One main concern here is time. There is no desire to recompile source files that have not been altered since they were last compiled. Therefore, time is spared if the dates of the last revision of the source and the corresponding object files can be compared. This same philosophy also applies for the overall executable: why relink the object files if no changes have occurred to any object file since the last linking. Most operating systems provide facilities for partially automated builds in the form of build command files. Examples of these include the *make* tool in UNIX [5, 24] and *MMS* in VMS [26]. Therefore, using the dependency information within the repository, it is possible to generate either partial or complete makefiles to perform application executable file builds. As another option, a manual build can be performed piece by piece based upon the contents

of the repository. Considering that so many tools already automate this for you, there seems no need to tackle the problem manually.

Some tools provide more advanced functions as well. One useful functionality that an automated build tool may provide is that of resolving the problem of different files within a configuration residing in different directories or possibly on different host machines. Another would be to provide support for parallel builds [2, 12].

3.2.3 Release Management

Once a configuration has attained a certain level of approval, plans are usually made to release for use by the user community. Prior to releasing, it must be verified that all constituent components within the configuration have attained the highest level of acceptance. Also, the executable file for the approved configuration must be built. Although not essential, a release management service would be a very useful component in a commercial application development organization.

There are a some issues to consider when releasing a particular application version. First, what items are to be included in the release: usually the executable program file and the user and system documentation. In addition, a great deal of information of use to the potential users must be provided. This is usually included in what is referred to as release notes. Among other things, described in the release notes would be which item versions are included in the application, what the system demands of the application are, how to perform the installation properly, how to upgrade to the new release, what the faults and limitations in the released system are, what the differences between this release and the previous one are, and how faults in the system are to be reported. Of these questions, two are easily answerable using the contents of the repository: which versions are included within

the application configuration version and what are the differences between the current and previous releases.

3.2.4 Project Management

Since the repository manages all of the information assets of an organization, it is the ideal environment to implement a project management system. Representing the business side of software development, project management encompasses two primary activities: project planning and project control [16]. Project planning involves the defining of the resources, tasks, procedures, and timings of a project. Project control involves the tracking of the progress of the project including its quality, resource utilization, and direction.

There are seven items the must be defined in the creation of a project model. Each of these definition steps should be supported by the project management system [16].

1. **Project Structure**. The project management system should support the representation of the tasks resulting from the decomposition of a project. For each task, individual ownership and time frame must be clearly definable.

2. **Activities**. The project management system should support the representation of all design activities required to design and complete all project tasks.

3. **Relationships**. The project management system should support the representation of time dependencies between activities.

4. **External Dependencies**. Resources and deliverables that are external but required for the project must be representable.

5. **Process Structure**. The project management system should enable the representation of a logical process sequencing of activities.

6. **Resource and Time Frames**. Required resources (people, equipment, software, funds) should be identified and assigned to activities. Further, a time span will need to be assigned to all activities based upon resource availability.

7. **Resource requirements**. The amounts of each resource required will be estimated and noted.

As a project progresses, time frames may change and new details may become available. Therefore, the contents of the project management system must be easily alterable.

A project management service benefits both managers and developers. Managers will use this service to define and control all active projects in a development organization. Developers will use this service to view current projects, tasks, and deadlines. Although this service is not essential in a repository environment, it can be quiet beneficial in taking information and human resource management to a higher, more structured and business oriented level.

A subset of the project management scheme is essential within a repository environment. Since the components that compose a configuration can originate from and apply to many distinct tools and subsystems, the relationship between a configuration and its components cannot be defined automatically through basic repository population techniques. Therefore, the repository should provide a user level interface to define the components within a configuration in addition to the configuration bind indicator if required. This will also provide a route to pass important application level metadata into the repository.

3.3 Impact Analysis

In a highly integrated computing environment, there can exist many dependencies between the different types of information resources. For example, one code module may depend on a table in a specific relational database having a specific structure. Any change in the structure of this table will require an accompanying change in the code module. If an organization specializes in database application development (or other highly integrated development), there will be a standing need to determine what impact the changing of certain software items will have on other existing software items. Therefore, it is crucial to be able to assess the impact a change to one item may have to other items. Impact analysis services expose the true value of the repository's integrated nature and therefore are a desirable functionality to provide.

In the change review stage, it is necessary to examine the cost and impact of a proposed change. Due to the highly integrated nature of software items in modern computing environments, it is likely that a change to one item may adversely affect many other software items. For example, a change to a database table or to a code module may make it necessary to make changes to related database views and models or related applications, respectively. A change could produce an adverse domino like effect upon many other software items and systems. This cascading effect is quite difficult to track manually. For this reason, providing the support for impact analysis for a given change is extremely useful as well as essential in the application development realm.

Since the repository is structured to maintain all of the necessary dependency information about key managed software items, it is not a difficult task to expose which items a proposed change might effect. The primary output of an impact analysis service is an

impact report. This report will indicate the specific software items that are potentially affected by a proposed change to another software item. For the user contemplating the changes, the impact analysis report can be viewed as a component in the overall feasibility study of enacting the change. It should be mentioned, however, that simply because one item is dependent upon the changed item does not mean that the change will require a corresponding change in the effected item: it merely warns of the potential for revision. Each affected developer would have to further analyze the change and determine what level of revision will be necessary if any–for example, relinking, recompilation, and/or recoding.

3.3.1 User Notification

It is necessary to be able to trace the effect of a given change to the affected user community and provide appropriate notification if the change is actually enacted. This notification can be implemented in several ways: by attaching notification messages to tools or applications or sending electronic mail to the affected users. If attached to tools or applications, the message will be displayed to a user upon running the tool or application.

As long as system user information is represented and tracked within the repository, this task is not difficult to implement. This notification report should include specifics about the change and possibly a reference to the corresponding change request, if any, so that the affected users can determine whether the enacted change will require change to the affected item without having to delve into the changed item itself.

3.4 Type Management/Transformation

This service is focused on the needs of the developer. The key idea here is to keep track of definitional information similar to that found in a typical C/C++ header file [15]. To

enable tracking at this low level, a microscopic view of the application must be taken. That is, instead of focusing on the file or module as the base application object, we must go one level down: data type and structural information must be represented. Moreover, the relationships between the definitional components as well as their relationships to their parent files or modules must be maintained.

There are two primary advantages to this approach. First, since many structures, routines, and classes are used by many developers for different projects, maintaining a library of all of these definitions can allow for maximal code reuse and automatic code generation [20]. Second, if header files can be eliminated entirely and automatic code insertion of needed structural definitions and data types at compilation time is implemented, development time can be sped up. Moreover, since only the definitions required for a specific module are included, the usual overhead of inserting entire header files into the code, even if not all of the definitions are needed, is reduced. Consequentially, the underlying complexity of the repository as well as the overhead of maintaining the relationships is increased when implementing at this depth.

In implementing such a service, it is necessary to have sophisticated source code parsers for extracting the desired definitional information. This service is not an essential participant in an end-user oriented repository package. However, it could be very useful in a strict application development arena.

3.5 Modification Tracking

The maintaining of audit trails can aid in monitoring the access to stored data, dealing with access violations, monitoring authorization levels, and performing verification and validation

of data. Modification tracking revolves around tracking information about changes made to information resources. In the case of changes within a DBMS, most of the information is already recorded in the DBMS log files. In the case of an application, modification history data may be tracked by a version or configuration system. Therefore, one can implement a modification tracking service by accumulating already existing log information into the repository in an acceptable form.

Selecting the appropriate scope of the modification tracking implementation is a crucial design decision. Consider the situation of implementing a service to track modifications for a DBMS. We can view two levels of modification tracking when dealing with changes to a database. The more broad approach focuses only upon the definitional changes, or rather changes to tables or table structure within a database. For instance, the who, what, and when of creating, altering, and dropping a table is tracked. At a lower, more detailed level, we can track instance changes, or rather the who, what, and when of inserts, deletes, and updates to the tuples within a table. The second approach requires much memory and may be infeasible for many organizations. Similar scope of tracking concerns exist for the tracking of other information resources.

3.6 Usage Analysis

Another key area that the repository can actively participate in is that of gathering statistical information about different niches of the computing environment. The information gathered can be useful to users, developers, managers, and system administrators.

Usage analysis services reveal the true management capabilities of a repository in providing information for the users that can be used to increase their productivity as well as

reduce inefficient hardware usage. One of the primary objectives of usage analysis is to determine the value of information and applications by knowing primarily, how often they are used and secondarily, who uses them. This information can be traced by maintaining access relationships between resources (e.g., between users and applications) on a frequency of use or access level. Knowing the relative importance of an application lets an organization know where to focus their maintenance and improvement efforts for maximal effectiveness. Further, from this analysis, it is usually possible to improve or replace applications that are most heavily used. Such analysis is not solely limited to application software. A sophisticated DBMS provides a good example of specialized usage analysis. If tracking is done to determine which fields in database tables are frequently searched upon, information about where indexes are most needed can be obtained.

On the hardware side, it is crucial to be able to identify what is causing system overloads so as to improve system performance. In addition, this information can be helpful in isolating bottlenecks in the system that can in turn be reduced or eliminated to improve performance. It may also be possible to identify the most common sources of system failures. Such information is crucial to getting the highest productivity out of an organization's information systems.

Obtaining the usage analysis data is no trivial task, however. There are two primary routes of obtaining this data. First, resource to resource (e.g., user to application) interactions (e.g., user executes an application) can trigger a side-effect transaction that in turn updates the usage statistics of the repository. This approach may entail a high overhead, however. To reduce this overhead, usage analysis tracking could alternatively be enacted only for specific crucial application areas.

Much of system-specific usage information is already tracked by other systems such as

DBMS, OS, and configuration management systems. Therefore it may only be necessary to either compile or simply reference the log files of these other systems in the repository.

Further, it may not be necessary to track this information continuously. Samples can be taken during certain time periods to perform analysis upon. For instance, for a given day, all log file information for participating subsystems can be accumulated and transferred into the repository at the end of the day. Alternatively, all specified resource to resource interaction can be tracked as long as a configurable environment variable is set [15]. The depth of usage analysis information tracking will solely depend upon the needs of a particular organization.

3.7 User Authorization

The repository maintains detailed information on all information assets of an organization. It is likely that some of the information maintained may be of a sensitive nature and thus, it would not be wise to allow users unrestricted access to the entire repository contents. Therefore, a flexible security mechanism should be implemented that will allow access to the repository to be restricted. There are many issues that must be considered.

User authorization must be addressed for two general categories of data: the contents of the repository itself and the external information resources that are referenced by the repository. Delineations in scope and type of access can also be explored. Type of access can be simplified as falling into three categories: read access, write access, or execute access. For any of these types of access there is an associated scope. For any given object in the repository a user can either have no access privileges, access privileges to a subset of the object (projection), or access to the entire object.

Access to Resources Referenced by the Repository

The information resources that the repository accesses will typically have their own access security policies defined. For example, both DBMS and operating systems generally have their own method of access control in place. Moreover, access to applications can be controlled by operating system level security policies. It may be desirable, however, to import the user level restrictions from these interfaced systems into the repository so that centralized security reporting capabilities can be easily implemented [15].

Access to the Repository Contents

What is being managed in the repository contents level of the user authorization process is a relationship between a user (or user group) and an object (or collection of objects) within the repository. There are several factors that can be used to determine whether a user can access a particular repository object: the object type, the project level affiliation of the object, and the permissions assigned to the user [21].

The repository architecture opens itself to three main security levels wherein user authorization can be implemented [21].

1. **Repository Level**. At this level, a user may be granted "all or nothing" access to the contents of the repository. The type of access (read, write, or execute) can be tailorable per user.

2. **Tool Level**. At this level, each interfacing tool is assigned a security access level. Certain tools, for instance, would be allowed read-only access to the repository contents. In turn, any user attempting to access repository data through this tool is restricted to read-only access as well.

3. **Metadata Level**. At this level, access is restricted at the level of the object (or collection of objects). Security at this level can be divided into three sublevels: meta-model security focuses on access restriction to a collection of related objects and their corresponding data instances (e.g., at the submodel level), metadata security focuses on access restriction to individual objects (e.g,. at the table level), and instance data security focuses on access restriction to instances of objects (e.g,. at the row level). Implementation at the instance level is the most difficult to implement and manage.

Oracle Version 7 provides a flexible user authorization mechanism [11]. In this approach, each user is assigned a role that indicates the access privileges of the user. These privileges are split into two main types: system privileges and object privileges. System privileges pertain to actions at the definitional level. For example, this could include access to perform table definition creation, alteration, and dropping. Object privileges, on the other hand, deal with access at the instance level. This could encompass record inserts, deletes, and updates.

This approach can be extended into the repository realm by assigning similar user privileges to tools interfaced with the repository as well as the users [15]. When users access the repository contents in conjunction with interfaced tools either an intersection of the role privileges or the minimal access level of the pair will determine the level of access.

3.8 Subscription Services

Being the centralized manager of information about all of an organization's information resources, the repository has a vast amount of valuable information that can be provided to the user community. Such information can range from project level status information

to system and information resource usage statistics information. This information can be compiled into reports and distributed electronically to users on a monthly, weekly, or daily basis.

For example, a project manager depends upon knowing the status of each component of every project that they oversee. Therefore, a project status report generated at the end of each working day that details the status of all components of specific projects would be invaluable for the manager. This report could include the acceptance level (e.g., *working*, *unit tested*, *system tested*, *approved*) attained by each module and document associated with an application configuration, which of these components are checked-out to a user and who that user is, and the date each component was last modified.

It would not be reasonable to send all generated reports to all users (e.g., end-users, developers, managers, system administrators). End-users would have little need for project status nor system usage reports. Each of these users will have special interests that should be addressed. Therefore, the subscription service should be configurable in that only certain user groups or individual users receive certain reports. Further, a user should be able to specify the frequency at which a report is distributed to them. These user profiles should be easily updated and maintained.

Chapter 4

Design and Implementation Issues

In an effort to evaluate and analyze the complexity of the techniques in designing and implementing repository services, a collection of basic repository services was developed. This collection of services was not intended to be all encompassing and complete but rather a prototype implementation.

To better grasp the scope and background of this effort, it is necessary to explore the customer requirements that directed the design of the underlying repository. The prototype repository was designed with the needs of Westinghouse Savannah River Company (WSRC) in mind. WSRC is not a commercial software development organization and is therefore not interested in the repository as a CASE tool integrator nor a structured application development environment provider. Their primary aim is to use repository technology to meet the needs of their vast and diverse end-user community.

The meta model of the prototype repository is limited to some extent by the scope of these requirements. The prototype repository is described in [15]. Since the services addressed herein deal primarily with the database and application subsystems of the repository, their data submodels are presented in Appendix A. Additionally, the physical database

table definitions for all repository tables relevant to the RADS system are presented in Appendix B.

The base program encompassing this collection of services was written using the C programming language with embedded SQL using Oracle's Pro*C Release 1.5.9.0.1 on a UNIX system running SunOS Release 4.1.3. Additional small external portions of the program were written in UNIX C-shell scripts and the C programming language. The database for the repository was implemented using Oracle Version 7, a relational DBMS.

Although the prototype repository itself was developed based upon the requirements of WSRC, the design of the collection of repository services was intended to be more general in nature. The primary aim of these services is to aid in the application development process. Accordingly, this collection of services will be referred to as Repository-based Application Development Services or RADS hereafter.

Included in the RADS system is support for version management, configuration management, impact analysis, project management, and a small subset of user authorization. The following sections will detail important design and implementation issues surrounding the development of the prototype RADS system. Details on the usage of the RADS system are presented in Appendix D.

4.1 Version Management

A strict version management mechanism is provided in RADS in which all managed items are explicitly versioned within the repository. Simply providing version management of externally versioned items is somewhat trivial as a repository service. Therefore, to analyze and gain better insight into the issues of developing a version control system as a repository service, a base collection of objects, both atomic and composite, were selected to be under direct version control of the repository. All pertinent version management design and

implementation issues are covered in the following sections.

4.1.1 Versionable Items

As a result of the structure of the meta model of the repository, only a small subset of the objects managed by the repository are versionable. Among the versionable items are code and definitional modules, documents, and application configurations.

4.1.2 Identifying Versions

The item versions managed by the version management service are identified using the purely numeric three level versioning scheme described in Section 3.1.3. This is facilitated by maintaining an item version table for each versionable software item that contains release, revision, and variant number attributes. In addition, a version history table is maintained for each versioned item that stores predecessor to successor version relationships. Through this pairing, the version identification scheme is fully represented.

There is one inherent limitation in this structure: all versioning must follow the same version identification scheme. A better choice would have been to have a single attribute of character string type in the item version table represent the entire version identifier. This generic approach would allow for identifiers in any versioning scheme to be represented. If a generic scheme were used, the versioning of items under the complete control of the repository would be closely regulated and integrity checked. However, for items explicitly versioned externally of the repository, the versioning scheme would only be supported in that the version identifiers for each software items as well as their predecessor to successor relationships would be maintained.

4.1.3 Tracking Change for Simple Items

Software items are not tracked by version identifier alone. To supplement version information an acceptance level indicator is maintained for each item version. This will let the developer know what level of acceptance or testing a specific item version has attained.

For modules and application versions, acceptance levels of *working*, *unit tested*, *system tested*, and *approved* are permitted. For documents, acceptance levels of *draft*, *under review*, and *approved* are supported. This is simply represented as an attribute in the item version table. The use of acceptance levels is optional. However, for the proper handling of the incrementing of application configuration version identifiers, it is strongly recommended.

All but the item instances at the leaves of the version history tree are frozen from further development. That is, once an item has been checked-in after revision, its contents are permanently stored and are unchangable. This provides an immutable history of all revisions to a specific software item. This is accomplished in two ways. First, the check-out facility prohibits the editable check-out of non-terminal item versions. Additionally, these versions are protected from revision by having their UNIX system file permissions set to a read-only mode. Note that it is possible to create a branch from a frozen item version upon which revisions can be performed.

4.1.4 Revising Items

Item revisions are controlled by a copy-in/copy-out implementation of the check-in/check-out scheme. In this implementation, all managed items (including all item instances) are stored within a centralized library controlled by the RADS system [26]. Refer to Section 4.1.7 for details on the structure of the library. This change control mechanism has two primary operations: check-out and check-in. Each of these operations will be detailed below.

Check-Out Operation

Two types of check-outs are supported: editable and read-only. Editable check-outs are provided for atomic items only (modules and documents) whereas both atomic and composite read-only check-outs are supported. The design choice to not allow editable configuration check-outs was primarily to simplify the development effort. There is no inherent obstacle to implementing check-outs at this level, however.

In checking out an item version, several constraints are imposed. First, the item version to be checked-out must be clearly identified by item name, item type, and item version identifier number. Next, a check is done to verify that the user has permission to check-out the item version. This user verification is done on two levels: a user is permitted to check-out an item if they are either the assigned owner of the item version or a member of one of the development or maintenance teams associated with an application configuration that includes the item. Refer to Section 4.4 for details on the user authorization process. Next, a check is done to verify that the version is not currently checked-out to another user (unless a read-only check-out is being attempted). Last, the item version is checked to determine if it is a terminal version. If it is not (i.e., it is frozen) an editable check-out will fail.

If all of these conditions are met, the check-out proceeds. The source file for the item version is recreated (see Section 4.1.6) and copied into the user's current directory with UNIX write permissions set. As a result, the item instance is marked as checked-out in the item version table. Note that read-only check-outs are simple source code recreations—the item version is not marked as checked-out. In the case of a configuration checkout, all associated document or module versions are recreated and copied into the user's current directory.

Check-In Operation

There are two types of check-ins supported: reserved and unreserved. On an unreserved check-in, the item instance is marked as checked-in but the revisions are not recorded in the library. For both types of check-ins, several restrictions are imposed. First, a unique identification of the item to be checked-in based upon item name, item type, and item version identifier must be made. Then, the user must be deemed eligible to check the item version in (see Section 4.4). Next, the item version must be currently marked as checked-out. Last, the file containing the item version must be present in the user's current directory. This file must have the same name as the item version identified to be checked-in.

If the check-in is unreserved, the item version is marked as checked-in and the check-in process is complete. However, if the check-in is reserved and all of these conditions are met, the process proceeds. An unreserved check-in is functionally the same as the un-check-out operation.

At this point, the user may change the acceptance level of the version. An automated upgrade/downgrade mechanism is provided based upon the sequence of acceptance levels presented in Section 4.1.3. If the acceptance level of the item version is higher than the lowest possible level, the version identifier is automatically incremented. However, if the acceptance level is at the lowest level, the user may opt to either hold the entire item version identifier at its current value or to increment it. If the user opts to increment the version identifier, one of two actions will occur. If the user is performing a check-in upon the main version branch, then the revision number component of the version identifier will be incremented. However, if the check-in occurs upon a variant branch, the variant number component of the version identifier will be incremented. The source file containing the item

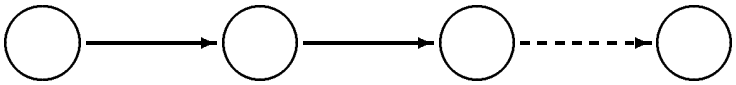

Figure 4.1: Check-in Process Illustration

version is then moved from the user's current working directory into the RADS library and the UNIX system file permissions are set to read-only. The user is required to supply a description of the changes made if new item version was created.

Three tables are affected by a reserved check-in. A new entry is added to the files table and the item version table. Then a linkage between the previous item version and the checked-in version is created in the item version history table. To illustrate this process more clearly, consider Figure 4.1. The user is performing a check-in of module *stack* version *2.2.0* and in turn, incrementing the revision number component of the version identifier. The new entry to the files table will represent the file *stack.2.3.0.c* which will contain the entire checked-in file. Since a new version of the *stack* module was created, an entry will be added to the module version table. Last, an entry is created in the module version history table linking *stack* version *2.2.0* to its new successor version *2.3.0*.

If the item being checked-in is a definitional module, an additional process to reconfigure any source to include file linkage relationships is performed. Within an application configuration, both code and definitional modules may be either statically or dynamically contained. This single level containment relationship implies nothing about the relationship between the code and definitional modules themselves, however. For example, if three code modules and one definitional module participate in the same application configuration, it is not assumed that all three code modules depend upon the single definitional module. (This

information will either be explicitly defined by the user in the project management process or extracted from **#***include* directives during the check-in process.) Further, the code to definitional module dependency relationship is defined on a static file to file basis. It would be unwise to make the assumption that a particular source code module will always contain the most recent version of a definitional module since there is no notion of a static version dynamic relationship here. However, it is not reasonable to expect the user to continually update these relationships manually each time a new definitional module version is created.

As a compromise, the automatic reconfiguring of code to definitional module relationships will only occur within the context of an application configuration: only if the definitional module and a code module are both dynamically contained within the same application configuration version and a source to include relationship is defined between the two will an automatic reconfiguring occur. This operation proceeds as follows. For all terminal application versions that dynamically contain the definitional module, a listing of all other module versions that participate in a source file to include file relationship with the definitional module is created. This relationship is updated to include this newly checked-in version of the definitional module instead of the previous one for each module in the list.

If the item being checked-in is a code module, however, a similar concern exists. Once a user explicitly defines a code module to definitional module version association, it is assumed that the assigned code module version as well as all of its successor versions will be associated with the definitional module. This relationship must be removed explicitly by the user to end the association. Therefore, any time a new code module version is created, all previously defined definitional module associations will also apply to the new version. Note that this update only occurs when the included file is explicitly defined within the repository as a definitional module—general include file relationships will not by default

apply to the new code module version.

Next, both code and definitional modules are scanned for general header files that they reference. This is performed by scanning the source file in question for any **#***include* directives. From this, the name of the file to be included is extracted and a match is attempted to be made between it and a valid file name in the files table. If exactly one such match is made, the relationship is noted in the source to include relationship table (if the item version is a code module) or in the include to include relationship table (if the item version is a definitional module). This process is very similar to that of the X-windowing system *makedepend* utility with one exception. Include to include dependencies are not recursively tracked here. It is assumed that all included files will have entries in the files table in the repository. Additionally, it is assumed that for each such include file, the include to include relationships that it participates in are already accurately stored within the repository.

Next, a crucial step in the configuration management process is performed. This entails a reconfiguration of all pertinent application configuration to atomic item dynamic configuration linkages. A list of all terminal application versions to which the document or module version being checked-in is dynamically configured is formed. For each entry in this list, the item version to application configuration version relationship is updated to include the new item version.

The item version is then marked as checked-in and delta storage is performed (see Section 4.1.6). If the acceptance level of the item checked-in was altered as a result in the check-in process, notification is sent to all affected users via the UNIX system electronic mail system. Affected users are defined as all members of the development and maintenance teams for any application configuration in which the item participates.

As a final step, if the acceptance level of the checked-in item has changed, the acceptance

level of all application configurations that the item version participates in is analyzed. In this process, a list of all terminal application versions to which the checked-in item is associated is gathered. For each of these, the maximal acceptance level attained by all of its participating module versions is determined. If this level is different than that which is currently assigned to the application version in question, a new application version will be created. New entries are created in the application version table, application version history table (to note the linkage between the old and new versions), and the files table (for the executable file for the new version). Since no change has occurred in the components of the configuration in this process, all application version to document version, module version, database object, and object library file relationships for the existing version are created for the new version as well. Further, notification noting the acceptance level change and new application version creation is sent to all affected user groups through the UNIX electronic mail system.

4.1.5 Concurrent Revisions/Merging

Standard check-in/check-out operations restrict check-outs to terminal (or non-frozen) item versions only. Therefore, in order to check-out a copy of a frozen version or a copy of a version that is already checked-out (parallel development), a branch must be created upon which to make revisions. Note that as a result of the limitations of the version identification scheme used, a branch can only originate from the main version line and only one branch can originate from any single item instance.

Implementing the creation of the branch is quite simple. The source item version where the branch will originate must be recreated (see Section 4.1.6) and copied into a new file. New entries in the item version and file tables are created for the new file. The version

identifier of the new item is generated by taking the version identifier of the source item version and replacing the variant component with "1". An item version history linkage is then created between the source and new item versions. Since the new version is now a terminal version, its contents are stored completely (i.e., it is not delta stored).

Merging, the reverse operation of branching, is supported as well. Three files must be identified and recreated: the two item versions to be merged and their nearest common ancestor. From this point, it must be determined whether a merging conflict exists. This is performed using the UNIX *diff3* command. If the merge is non-conflicting, then a merged file will be created. This file which will contain all of the revisions of both of the versions to be merged is generated by finding all line based textual revisions that must be applied to the ancestor to recreate the first item version. This set of changes is in the form of a sequence of UNIX *ed* commands. The *ed* utility is a basic text editor that accepts line based commands including add (a), delete (d), and change (c). The UNIX *diff* command is used to generate this set of *ed* commands. Last, the sequence of *ed* commands will be applied to the second item version to create the final merged file. Note that this merge technique is a line by line text file based merge. There is no support for merging non-text files.

Once a merge has been successfully performed and the user is content with the changes, a version history linkage from the item version on the branch to the item version on the main line must be recorded in the item version history table. This is not the only step, however. This link may adversely affect the delta storage in place. The item version on the branch will only need to be delta stored if it was a terminal version. If it was not a terminal version, it will already be delta stored and therefore, no additional delta storage application will be necessary.

If the item version on the main line is not a terminal version, then its source must be

recreated prior to performing the delta storage. Refer to Section 4.1.6 for details on the delta storage and recreation techniques. For the linking of application configuration version history, the link is purely historical and no delta storage is performed.

4.1.6 Version Storage Technique

All item instances are not stored in their entirety. Alternatively, only terminal versions are stored in their entirety and all other versions are stored as differences (or deltas) between consecutive versions. This is referred to as *reverse deltas* and is based upon the technique implemented in RCS [23]. Note that this form of reduced storage is only applicable for text files.

There are two operations required in this reduced space version storage mechanism: a delta storage operation and a source recreation operation. Upon checking-in an item version into the RADS system, it is guaranteed that the item version will have exactly one immediate predecessor version. This is so because on check-in, a single item is always linked to exactly one predecessor upon its same version branch. Check-ins resulting from merges are performed in a two step manner so that one version history link is formed at check-in and the other link must be explicitly defined by the user.

To perform delta storage, the UNIX *diff* command is used to generate the textual differences between the new terminal version and its immediate predecessor. This collection of differences, represented in the UNIX *ed* format, then replace the contents of the source file for the predecessor version. File permissions are altered accordingly in order to perform this operation.

Source recreations perform the reverse task. For this process, two versions are identified: the item version to be recreated and a terminal version of the same item. There may be

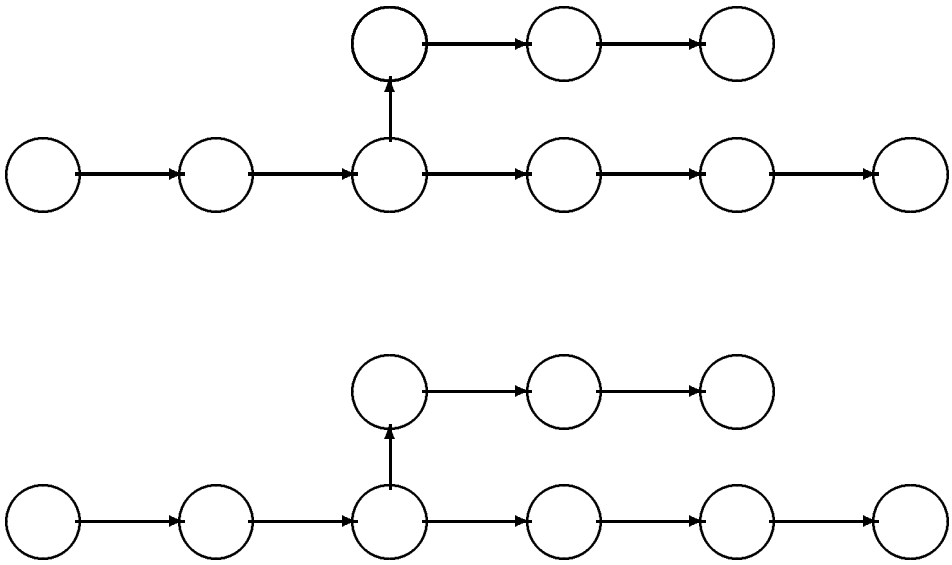

Figure 4.2: Terminal Version Selection for Source File Recreation

more than one potential choice for the terminal version selection. In this case, the terminal version on the most direct path to the item version to be recreated will be selected. For example, in Figure 4.2a if the version to be recreated is *1.2.2* then version *1.2.3* must be selected as the terminal version. However, in Figure 4.2b if the version to be recreated is *1.2.0* then version *1.5.0* will be selected as the terminal version.

In the course of progressing up the version tree from the terminal version to the selected version, the correct predecessor must be chosen. A predecessor on a branch will only be chosen over a predecessor on the main line if the version to be recreated lies directly on that branch. For example, in Figure 4.3a since the version to be recreated (*1.1.2*) is located on a variant branch, the route through version *1.1.3* will be chosen. On the other hand, if version *1.2.0* is to be recreated as in Figure 4.3b, the route along the main branch will be selected. The delta commands stored in each file will be recursively applied to the terminal version and all versions between it and the version to be recreated. For example, in the recreation

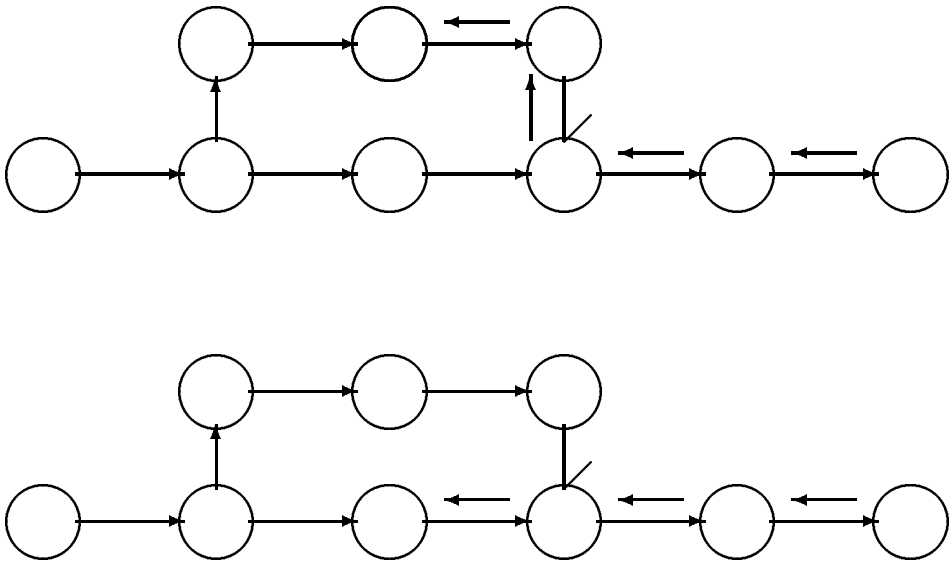

Figure 4.3: Predecessor Selection for Source File Recreation

of version *1.1.2* in Figure 4.3(a), the change command file for version *1.4.0* will be applied to the full text of version *1.5.0* to recreate version *1.4.0*. Next, the change command file for version *1.3.0* will be applied to the full text of version *1.4.0* to recreate version *1.3.0*. This process will continue until version *1.1.2* is recreated in its entirety. The application of the deltas at each step up the version tree is performed using the UNIX *ed* utility.

4.1.7 Library Structure

Copy-in/Copy-out implementations of the check-out/check-in model require that a library be established to control the versionable items while they are not checked-out to a user. This library is structured hierarchically in the RADS system within the UNIX directory structure. Figure 4.4 illustrates the library directory structure. There are five subdirectories located within a main directory that allow the types of managed files to be categorized. Module source files are stored in *SRC*, document files are stored in *DOC*, module object

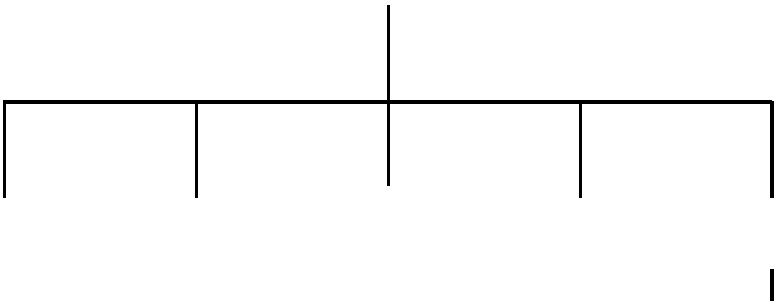

Figure 4.4: Structure of RADS Library

files are stored in *OBJ*, application program executable files in *BIN*, and release directories for each application configuration version released are in *REL*. This structuring is based upon a technique described in [26].

The *OBJ* subdirectory contains a complete object file for each code module version. It is not necessary to store all object files; therefore, although not currently implemented, it would be beneficial to define a protocol for periodically purging old files from this directory. There is one benefit to retaining these files, however. It may prevent the need for recompilation of source files if existing up to date object files are maintained. The *SRC* subdirectory contains a file for each module version and the *DOC* subdirectory contains a file for each document version, but only the files representing terminal versions are stored in their entirety. The rest of the source files contain only the UNIX *ed* commands required for their recreation. Refer to Appendix C for complete textual listings of a sequence of files in the SRC directory. The *BIN* subdirectory will contain one executable file per application configuration version. The files in all of these directories have the version identifier embedded in the file name to guarantee the uniqueness of the file name. Each time an application

configuration version is released, a new subdirectory whose name indicates the application and version will be created under *REL*. Contained in each of these individual release directories will be the executable program file (copied directly from the *BIN* subdirectory), all associated recreated documentation files, and a report that details all components of the application configuration version.

Direct access to the contents of the library is controlled by UNIX system file permissions. Users have no access privileges for *SRC*, write privileges for *OBJ* (for the creation of object files), execute and write privileges for *BIN* (for creating and running executable files), and read and execute privileges for *REL* and its contained subdirectories (for viewing documentation and configuration reports as well as running the executable).

4.1.8 Change Request System

The change request subsystem provides an automated approach to managing the change process. The key function in this subsystem is to manage the life cycle of the change requests. These change requests are stored within a stand-alone table within the repository database. A change request may go through many distinct phases in its lifetime. The RADS system supports four life cycle stages: *under review*, *approved*, *rejected*, and *implemented*.

Upon its creation, a change request must be associated with a distinct module version. Note that only change requests for modules are currently supported. A facility is provided which permits a user to easily track a change request through all stages of its life cycle. At the initiation of each stage, important information is retained including the user who was responsible for the status change (e.g., the originator, approver, or implementor) and the details and description of the change.

To enable easy access to the change request data, three specialized reports are provided.

One report will display all information about a single change request. The second will display all change requests associated with a particular module. The last provides a list of all outstanding change requests; that is, all change requests that are either *under review* or *approved* but not yet *implemented*.

4.1.9 Reports

Three primary reports are available within the RADS version management service: version history, change history, and catalog reports. Each of these reports is provided for modules, documents, and application configurations.

Version history reports provide an illustration of the structure of the version history tree for an item version. This report is generated by first collecting a list of all existing versions of a particular item. Then, for each version entry in the list, a list of immediate successor versions is gathered. This information is sufficient to completely describe the version history tree.

Change history reports are not concerned with the structure of the version tree but rather the changes enacted that resulted in each item version. For each existing version of an item, the version identifier, change description, and date of last modification are extracted. If the item is a module, then the change request table is checked to determine if the version resulted as a consequence of a change request. All of this information is in turn accumulated and displayed to the user.

Catalog reports are intended to provide the user with a listing of all available modules, documents, and applications. The primary benefit hoped to be gained from these is the identification of items for potential code reuse. Document and application catalog reports are simple. These are generated by scanning the base item table for a list of all defined

item names and their description. Two types of object catalog reports are provided for modules: a simple inventory report and a detailed functional level report. The generation of both reports progress the same initially. A list of all terminal module versions on main version history lines is created that includes the module name, terminal version identifier, description, and definitional component indicator. Next, if a detailed report is being generated, the source for the terminal module version is recreated and parsed for functional and data definitions. This is accomplished using a simple recursive descent parser that ignores comments and code blocks (for C/C++ files) while extracting everything else within a file. All of this information is then compiled into the module catalog report.

4.1.10 Commentary

Although the small collection of versionable items provided a good basis for evaluating design and implementation techniques and complexity, the scope of managed items would need to be extended for an active production environment. The version identification scheme used, though flexible and functionable, is too constricting to support the representation of version identifiers for both internal and external versioning schemes. Acceptance levels should ideally be configurable for an organization and not rigid as implemented herein. The check-in/check-out mechanism in RADS is quite adequate and left little to be desired. Concurrent revisions were supported adequately. Version storage and recreation techniques are tedious to implement but not unmanageable.

4.2 Configuration Management

The configuration management mechanism within the RADS system supports application realm specific configurations consisting of a grouping of two versionable component types

and one non-versionable component type. Modules and documents, both versioned at the atomic level, and object library files, which are not versioned, can be components within an application configuration.

The configurations themselves are explicitly versioned within the repository using the versioning scheme detailed in Section 4.1.2. In addition, each application configuration version will have an acceptance level indicator associated with it. The acceptance levels supported are described in Section 4.1.3. This acceptance level is not explicitly assigned to configurations as it is for atomic objects. The acceptance level of any application configuration version is equal to the highest acceptance level attained by all of its versioned components. This acceptance level is automatically updated upon the check-in of any component in the configuration wherein the acceptance level of the item was altered in the check-in process (see Section 4.1.4).

This process is not without processing overhead, however. Table 4.1 presents the results of the timing analysis of automatically updating acceptance level status values. This analysis involved the checking-in of a single module where (1) the acceptance level value of the module was unchanged and (2) the acceptance level value of the module was altered. This was examined where the module was a component of 1, 3, 5, and 10 application configurations wherein each configuration had a total of three components including the module version being checked-in. Each numeric result presented in the table was calculated by taking the average of at least three test runs each under the same set of conditions.

The containment relationship between any versionable configuration component and its parent configuration may be defined as either dynamic or static. If the relationship is static, a specific version of a item is bound to a specific version of an application configuration. This relationship is unchanged unless done so explicitly by the user. However, if the relationship

Table 4.1: Acceptance Level Update Timing Analysis

Automatic Acceptance Level Update Timing Analysis		
No. of Configurations	No Acceptance Level Change	Acceptance Level Change
1	0 ms	23 ms
3	0 ms	29 ms
5	0 ms	36 ms
10	0 ms	46 ms

Table 4.2: Link Reconfiguration Timing Analysis

Reconfiguration Execution Timing Analysis		
No. of Configurations	Static	Dynamic
1	12 ms	44 ms
3	13 ms	91 ms
5	14 ms	170 ms
10	17 ms	363 ms

is dynamic, the most current version of an item is always bound to the most current version of an application configuration. There is a substantial overhead involved in maintaining dynamic configuration containment linkages as opposed to static linkages. Table 4.2 presents the results of a timing analysis of automatically reconfiguring dynamic configuration links. This analysis involved the checking-in of a single module version where the module was (1) statically configured and (2) dynamically configured to all of 1, 3, 5, and 10 applications. Each numeric result presented in the table was calculated by taking the average of at least three test runs each under the same set of conditions. The only point where this overhead is incurred is at check-in time and when an atomic element is released since these are the only times when the version identifier of an atomic item may change. The automated process of reconfiguring dynamic links is presented in Section 4.1.4.

Application configurations can be checked-out on a read-only basis in the RADS system. The general check-in/check-out mechanism is described in Section 4.1.4.

The following sections will outline the subsidiary configuration mangement services. Included are an automated configuration build service, a release management service, and a project management service.

4.2.1 Building an Application Configuration

Due to the many items and relationships that must be known to perform an accurate application executable build, it is not easy for a developer to perform the task manually. Since the repository maintains all components and their version identifiers along with their relationship to the application configuration version, it is natural to automate the build process as a repository service.

The RADS system does not execute the build itself but rather determines all application components and dependencies in order to generate a UNIX *makefile*. The key advantage to this automated process is that the correct version of each component in a specific application configuration version is correctly identified. This goes a step beyond the UNIX *makedepend* utility in that the relationships between the application configuration executable and its object and object library files is determined as well as the source code to include dependencies.

To initiate the automated build, the user must uniquely identify an application version which is managed by the system. In addition, the user may specify the directory paths for the source files, object files, and executable files. The remainder of this process involves extracting component version and relationship information from the repository. This process proceeds as follows:

1. All application version information is extracted. Pertinent information includes executable file and host name, linker, and linker options.

2. All non-definitional module versions that participate in the application configuration version as well as the compiler and compiler option data for each is retrieved.

3. For each non-definitional module, a list of files that their compilation is dependent upon (such as include files) is generated. This process is recursively applied to each such include file to determine all of its dependencies as well. The information for this process is contained in two repository level tables—one that maintains source to include file relationships and one that maintains include to include file relationships.

4. Information about all object library files associated to the application executable file is extracted. Pertinent information includes base source file name, host name, compiler, and compiler options. For each source file, the files its compilation is dependent upon is determined recursively as in Step 3.

All of this information is in turn compiled into a *makefile*. The entries in the *makefile* are based upon three dependency types:

- The application executable file depends upon all object and object library files associated with it,
- Object files depend upon their parent source files and all include files the source is dependent upon, and
- Archived object library files depend upon their object, source, and all include files the source is dependent upon.

Note that since delta storage is used for all module source files, it is best that the application configuration is checked-out in read-only mode and the *makefile* source file path is set to the directory containing the recreated files. Also, an automated build will not

proceed unless all component modules are checked-in. This requires that the configuration version be in a stable state prior to building.

4.2.2 Release Management

At distinct points in the development cycle of an application program, the software will be released for use by the user community. It is crucial at these times to preserve the structure of the configuration by freezing it from further change. This is accomplished by creating a new version of the application configuration.

To initiate the release process, the user must distinctly identify a terminal version of an application configuration. As a result, several actions will occur. First, a release subdirectory is created for the application version. The executable file and all associated document versions are recreated then copied into this directory. Also placed in this directory is a generated report that identifies all configuration components that compose the application version.

To freeze the structure of the current application configuration version, a new version of the configuration is created. In this process, new entries are created in the files, application version, and application version history tables. The new application version is created with the exact structure of its predecessor by setting all module version, document version, and object library file to application version links the same for the new version as its predecessor. Last, a notification report is sent to all affected user groups using the UNIX electronic mail system.

4.2.3 Project Management

This service will permit a user to explicitly define the components of an application configuration. There are four primary components that may be associated with an application configuration: modules, documents, object library files, and database objects. Herein, associations to modules, documents, and object library files may be defined. Information on defining the associations of database objects to application configurations can be found in Section 4.2.4.

In the prototype repository, modules and documents are the only versionable components in an application configuration. In defining this association, the explicit version identifier of both the item and the application configuration must be provided. Further, the linkage must be specified as either static or dynamic. If the relationship will be dynamic, the terminal item version in either the main line or on a branch should be specified.

Since object library files are not explicitly versioned within the repository, their association to application configurations is simple. Only the distinct application version identifier and a simple file name for the object library file are required to form the association. Since the object library file is not versionable, there is no issue of static versus dynamic configuring.

A supplementary function provided within this service is the ability to associate definitional module versions to code module versions. This association is on a purely file to file basis (which is defined by specific module version identifiers) and therefore, no static versus dynamic configuration issues exist. As a code module undergoes revision and new versions are created, all definitional module associations to the code module are automatically propagated to the new version. However, as a definitional module undergoes revision,

these associations are only updated if both modules participate in a dynamic linkage to the same application configuration version. Refer to Section 4.1.4 for further details on the reconfiguring of dynamic linkages.

4.2.4 Database Object to Application Associations

The prototype repository provides data structures for storing relationship data between applications and database objects. Since in a database application it would be very time consuming and costly for a user to define all application configuration to database object references manually, RADS provides an automated means of extracting this information from source files. Note that in the repository meta model, the only relationship supported between the database and application realms is a high level association between application versions and database objects.

To initiate this process, the user must identify a distinct application configuration version. In the capturing of this relationship data, all application version to database object relationship data is purged for the application configuration version under consideration. For each module version that participates in this configuration version, the text of the source file is recreated then parsed for references to database objects. Next, for each referenced database object extracted from the source file, a name and object type match between the extracted object and a database object managed by the repository is attempted. If a match is made, the database object to application version relationship is established within the repository.

This extraction is accomplished using a recursive descent parser. In this process, a text file is parsed for standard SQL code blocks for references to tables or views. The SQL statements supported are *create*, *alter*, *drop*, *insert*, *select*, *update*, and *delete*. Since table

references provide the most valuable application to database object relationship information, the extraction mechanism supports only the extraction of direct references to tables and views.

4.2.5 Reports

It is essential for the developer and managers to be able to retrieve detailed information about any existing application configuration version. RADS provides two key application level reports: a simple configuration status report and a complete, detailed application report.

The configuration status report contains a listing of all module and document components in the configuration along with their associated acceptance level and checked-out indicators.

The complete application report is much more detailed than the status report. Information for this report is extracted in the same fashion as the information for the automated build process (see Section 4.2.1). At each step in this process, all related meta data for each configuration component is extracted for the report. This report also includes a listing of all members of the development and maintenance teams for the application.

4.2.6 Commentary

The idea of a configuration should be generalized and therefore not be specific to the application realm. Participants in a configuration should include other key software items (such as database objects) as well. The configuration bind type plays a major role in the configuration management process. Although the maintenance of support for dynamically configured links incurs a higher overhead than for statically configured links, the value of

its flexibility outweighs the costs. The approach of creating a new application configuration version only when the acceptance level of the configuration was altered resulted in a sufficient number of historical versions in the application life cycle and the maintenance overhead was not overwhelming. The acceptance levels should be user configurable, not be rigid as implemented herein, however.

4.3 Impact Analysis

The impact analysis service allows a user to determine the effect a change to a specific software item has on application configurations maintained by the repository. The repository supports many relationships that make this analysis possible including database object, module version, document version, object library file, and general file to application version relationships. Therefore, the impact of changing any of these software items on existing application versions is accessible.

To initiate this process, the user must identify the software item to be changed. Note that this item must be managed by the repository. RADS supports four categories of software items: database objects (including tables, views, synonyms, sequences, indexes, columns, and clusters), module versions, document versions, and general files. For the selected software item, a list of all terminal application versions that are potentially affected by the item is generated. If the user decides to initiate the change, change notification will be sent through the UNIX electronic mail system to the development and maintenance groups for all impacted application versions. Included in this notification will be the details of and reasons for the change.

4.3.1 Commentary

The scope of cross system relationships between software items should be extended to more detailed levels than implemented herein. For instance, database object relationships should not only be traceable to a particular application version but also to specific code module versions. Further, relationships between source and include files as well as include and include files should be used in the impact analysis phase. Although the basic dependency information maintained provides much valuable information for users, this information cannot reveal at what level the impact of a change will occur. This information would be invaluable to developers.

4.4 User Authorization

User authorization is implemented on a small scale in the application development realm. Users can be assigned to one of two roles within an application or project: a developer or a maintainer. This information is represented by a relationship between a valid user in the system and a defined application configuration.

It is only by either being explicitly listed as an owner of a configuration component or a participant in one of these roles for an application that contains a component that a user may gain read or write access to that component. Application components are further secured through UNIX system file permission settings.

Chapter 5

Summary

A repository alone cannot provide the information management value that organizations demand. Therefore, repository services are an essential component in the overall repository package. Repository services provide two types support: internal management support for the repository itself and user level support for both the application development process and system performance analysis. The specific services that an organization will require will depend primarily upon the nature and needs of that organization. The design of the underlying repository will play a crucial role as well in determining the implementation approach, scope, and functional potential of the services.

The repository service details presented herein coupled with the collection of prototype repository services developed provide a sound basis for the evaluation and analysis of the complexity of designing and implementing key repository services. Through this investigation, certain service design choices were determined to be desirable, adequate, and functional while others proved to be too limiting in scope for large scale implementation. Further, through the development of basic services, repository meta model design limitations were uncovered. One of the key observations is that repository services should be

designed in conjunction and cooperation with the design of the meta model. It is very limiting to design and implement services on top of an already existing meta-model. This effort also uncovered key complexity issues including points of excessive processing and storage overhead. Overall, in revealing both its design strengths and limitations, this prototype collection of services provides an adequate model from which to base future repository service designs.

Several enhancements and extensions could be made to the collection repository services implemented. First, the version management system tackles only the management of items directly controlled by the repository. This should be extended to support the management of externally controlled items as well. In order for the repository versioning services to be complete, the repository meta model should be extended to allow the versioning of all key software items instead of just modules, documents, and application configurations.

One key repository aim was neglected in the development of both the prototype repository and the collection of services: complete support for an integrated CASE tool environment. With respect to enhancing services to support this aim, low level interface (API) calls would need to be provided for CASE tool to repository item revision transactions. Although not implemented, inherent in the design of the repository is the potential for tracking configurations and elements across different environments and machines. This is a valuable characteristic of the repository that should be explored.

Bibliography

[1] American National Standards Institute, Inc. *Information Resource System (IRDS) Standard X3.138-1988*. ANSI, New York, 1989.

[2] Atria Software, Inc., 1994. *ClearCase User's Manual*, May 1994.

[3] R. G. G. Cattell. *Object Data Management*, Addison-Wesley, Reading, Massachusetts, 1991.

[4] Electronic Industries Association. *CDIF - Standardized CASE Interchange Meta-Model (IS-83)*, EIA, Washington DC, 1991.

[5] Stuart I. Feldman. "Make—A Program for Maintaining Computer Programs", *Software Practice and Experience*, 9(3), pp. 255-265, March 1979.

[6] Per O. Flaaten, Donald J. McCubbrey, et. al. *Foundations of Business Systems*, Second Edition, Anderson Consulting, The Dryden Press, Orlando, Florida, 1992.

[7] Lilian Hobbs and Ken England. *Digital's CDD/Repository*, Digital Press, 1993.

[8] Cheng Hsu, M'hamed Bouziane, Laurie Rattner, and Lester Yee. "Information Resources Management in Heterogeneous, Distributed Environments: A Metadatabase

Approach," *IEEE Transactions on Software Engineering*, Vol. 17, No. 6, pp.604-625, June 1991.

[9] IBM. *Repository Manager/MVS Release 2: Guide for Users and Administrators*, IBM, March 1991.

[10] Warren Keuffel. "Paradigms in Configuration Management", *Software Development*, September 1993, pp. 23-28.

[11] George Kich and Robert Muller. *ORACLE 7: The Complete Reference*, Osborne McGraw-Hill, Berkeley, CA, 1993.

[12] David B. Leblang and Robert B. Chase, Jr. "Parallel Software Configuration Management in a Network Environment," *IEEE Software*, Vol. 4, No. 6, pp. 28-35, November 1987.

[13] Carma McClure. *The Three Rs of Software Engineering: Re-engineering, Repository, Reusability*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[14] V. J. Mecurio, B. F. Meyers, et. al. "AD/Cycle Strategy and Architecture," *IBM Systems Journal*, Vol. 29, No. 3, pp. 170-188, 1990.

[15] John A. Miller, Walter D. Potter, Krys J. Kochut, et. al. *Design of a WSRC Repository with an End-User Emphasis*, WSRC Technical Report, January 1994.

[16] T. Raz. "Introduction of the Project Management Discipline in a Software Development Organization", *IBM Systems Journal*, Vol. 32, No. 2, pp. 265-277, 1993.

[17] Marc J. Rochkind. "The Source Code Control System", *IEEE Transactions on Software Engineering*, SE-1(4), 364-370, December 1975.

[18] Alan R. Simon. *The Integrated CASE Tools Handbook*, Multiscience Press, Inc., New York, 1993.

[19] Ian Sommerville. *Software Engineering*, Addison-Wesley Publishing Company, Reading, MA, 1992.

[20] Bjarne Stroustrup. *The C++ Programming Language*, Second Edition, Addison-Wesley Publishing Company, Reading, MA, 1993.

[21] Adriene Tannenbaum. *Implementing a Corporate Repository: The Models Meet Reality*, John Wiley and Sons, New York, NY, 1994.

[22] Adriene Tannenbaum. "U.S. Market Seeing Repository Rebirth", *Software Magazine*, June 1993, pp. 43-61.

[23] Walter F. Tichy. "RCS—A System For Version Control", *Software–Practice and Experience*, 15(7), 637-654, July 1985.

[24] *UNIX System V Release 4 Programmer's Reference Manual*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1990.

[25] Lois Wakeman and Jonathan Jowett. *PCTE: The Standard for Open Repositories*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1993.

[26] David Whitgift. *Methods and Tools for Software Configuration Management*, John Wiley and Sons Ltd., 1991.

Appendix A

Meta Model Diagrams

In this section, extended entity relationship (EER) data models are presented for pertinent portions of the prototype repository meta model. Figure A.1 is the EER diagram for the applications life cycle system model, Figure A.2 is the EER diagram for the database system model, and Figure A.3 presents the overall repository meta model that relates these two systems together. The complete set of repository data model diagrams can be found in [15].

Figure A.1: Application Life Cycle Submodel

Figure A.2: Database System Submodel

Figure A.3: Repository Meta Model

Appendix B

Sample RADS Library Text Files

```
/*************************************************************************/
/************   Bonnie M. Edwards                              *********/
/************   CS 872 Project, Fall 1993                      *********/
/************                                                  *********/
/************   STACK.C   -----    Stack Primitive Functions   *********/
/*************************************************************************/

#include "hpc.h"


/** Return a non-zero number if the stack is empty and zero otherwise **/

int is_empty (STACK s)
{
   return (!s.count);
}


void create_stack (STACK *s)
{
   s->count = 0;
}


void push (STACK *s, TNODEPTR newitem)
{
   s->items[s->count] = newitem;
   (s->count)++;
}


void pop (STACK *s)
{
if (s->count == 0)
   printf("EMPTY STACK ENCOUNTERED!!!!!!!!\n");
(s->count)--;
}



TNODEPTR stacktop (STACK s, TNODEPTR *item)
{
   *item = s.items[s.count - 1];
   return (*item);
}
```

Figure B.1: stack.1.2.0.c

```
36,38c
if (s->count == 0)  printf("EMPTY STACK ENCOUNTERED!!!!!!!!\n");
   (s->count)--;
.
33a

.
26a

.
20a

.
12,14d
4a
/*************                                          *********/
.
```

Figure B.2: stack.1.1.0.c

```
1,49d
```

Figure B.3: stack.1.0.0.c

Appendix C

Subset of Repository Database Schema

TABLE: APPLICATIONS

Name	Null?	Type
APPL_NAME	NOT NULL	CHAR(30)
IDENTIFIER	NOT NULL	NUMBER(5)
APPL_GROUP_NAME	NOT NULL	CHAR(30)
PROCESS_MODEL_NAME	NOT NULL	CHAR(20)
PROCESS_MODEL_ID	NOT NULL	NUMBER(10)
APPL_GROUP_HOSTNAME	NOT NULL	CHAR(20)
BUSINESS_UNIT_ID		NUMBER(3)
SELECTOR_ID		NUMBER(5)
SCREEN_ID		NUMBER(5)
BASE_SCREEN_ID		NUMBER(5)
SELECTOR_NAME		CHAR(40)
CATEGORY		CHAR(30)
INVOCATION_STRING		CHAR(30)
OUTPUT_DESC		CHAR(80)
LONG_DESC		CHAR(240)
SUB_TYPE		CHAR(4)
TYPE		CHAR(30)
SHORT_DESC		CHAR(80)
PARAMETER_DESC		CHAR(20)
LOCATION_OF_MANUALS		CHAR(80)
INPUT_DESC		CHAR(80)
CREATION_DATE		DATE

TABLE: APPLICATION_DEVELOPED_BY

Name	Null?	Type
APPL_ID	NOT NULL	NUMBER(5)
USER_ID	NOT NULL	NUMBER(10)
ASSIGN_DATE	NOT NULL	DATE

TABLE: APPLICATION_MAINTAINED_BY

Name	Null?	Type
APPL_ID	NOT NULL	NUMBER(5)
USER_ID	NOT NULL	NUMBER(10)
ASSIGN_DATE	NOT NULL	DATE

TABLE: APPLICATION_VERSIONS

Name	Null?	Type
APPL_ID	NOT NULL	NUMBER(5)
APPL_RELEASE_NUM	NOT NULL	NUMBER(3)
APPL_REVISION_NUM	NOT NULL	NUMBER(3)
APPL_VARIANT_NUM	NOT NULL	NUMBER(3)
CREATION_DATE	NOT NULL	DATE
APPL_NAME	NOT NULL	CHAR(30)
CHECKED_OUT	NOT NULL	CHAR(1)
EXECUTABLE_MACHINE_ID		NUMBER(5)
EXECUTABLE_FILE_ID		NUMBER(6)
LAST_MODI_DATE		DATE
STATUS		CHAR(30)
LINKER		CHAR(20)
LINKER_OPTIONS		CHAR(30)
VARIANT_TYPE		CHAR(30)
DESCRIPTION		CHAR(50)

TABLE: APPLVER_DB_OBJECT

Name	Null?	Type
APPL_ID	NOT NULL	NUMBER(5)
APPL_RELEASE_NUM	NOT NULL	NUMBER(3)
APPL_REVISION_NUM	NOT NULL	NUMBER(3)
APPL_VARIANT_NUM	NOT NULL	NUMBER(3)
DB_OBJECT_ID	NOT NULL	NUMBER(6)

TABLE: APPL_VERSION_HISTORY

Name	Null?	Type
APPL_IDENTIFIER	NOT NULL	NUMBER(5)
APPL_PRED_RELEASE_NUM	NOT NULL	NUMBER(3)
APPL_PRED_REVISION_NUM	NOT NULL	NUMBER(3)
APPL_PRED_VARIANT_NUM	NOT NULL	NUMBER(3)
APPL_SUCC_RELEASE_NUM	NOT NULL	NUMBER(3)
APPL_SUCC_REVISION_NUM	NOT NULL	NUMBER(3)
APPL_SUCC_VARIANT_NUM	NOT NULL	NUMBER(3)

TABLE: APPL_VER_FILE

Name	Null?	Type
FILE_MACHINE_ID	NOT NULL	NUMBER(5)
FILE_IDENTIFIER	NOT NULL	NUMBER(6)
APPL_ID	NOT NULL	NUMBER(5)
APPL_RELEASE_NUM	NOT NULL	NUMBER(3)
APPL_REVISION_NUM	NOT NULL	NUMBER(3)
APPL_VARIANT_NUM	NOT NULL	NUMBER(3)

TABLE: CHANGE_REQUEST

Name	Null?	Type
CR_IDENTIFIER	NOT NULL	NUMBER(5)
CR_MODU_IDENTIFIER	NOT NULL	NUMBER(5)
CR_MODU_RELEASE_NUM	NOT NULL	NUMBER(3)
CR_MODU_REVISION_NUM	NOT NULL	NUMBER(3)
CR_MODU_VARIANT_NUM	NOT NULL	NUMBER(3)
CR_DESCRIPTION	NOT NULL	CHAR(240)
CR_REASON_DESC	NOT NULL	CHAR(240)
CR_PRIORITY		CHAR(40)
CR_ORIGINATOR	NOT NULL	NUMBER(10)
CR_ORIG_DATE	NOT NULL	DATE
CR_STATUS	NOT NULL	CHAR(30)
CR_APPROVER		NUMBER(10)
CR_APPR_DATE		DATE
CR_COST		NUMBER(4)
CR_APPR_COMMENTS		CHAR(120)
CR_IMPL_DESCRIPTION		CHAR(240)
CR_IMPL_COMMENTS		CHAR(120)
CR_IMPL_DATE		DATE
CR_NEW_MODU_RELEASE_NUM		NUMBER(3)
CR_NEW_MODU_REVISION_NUM		NUMBER(3)
CR_NEW_MODU_VARIANT_NUM		NUMBER(3)

TABLE: DATABASE_OBJECT

Name	Null?	Type
DB_OBJ_IDENTIFIER	NOT NULL	NUMBER(5)
DB_OBJ_NAME	NOT NULL	CHAR(20)
CREATION_DATE	NOT NULL	DATE
LAST_MODI_DATE	NOT NULL	DATE
SCHEMA_IDENTIFIER	NOT NULL	NUMBER(4)
TABLESPACE_IDENTIFIER	NOT NULL	NUMBER(5)
OBJECT_PRIV_IDENTIFIER		NUMBER(6)
OBJECT_TYPE	NOT NULL	NUMBER(3)
DESCRIPTION		CHAR(50)
DB_TYPE		CHAR(10)

TABLE: DOCUMENTS

Name	Null?	Type
DOCU_IDENTIFIER	NOT NULL	NUMBER(5)
DOCU_NAME	NOT NULL	CHAR(30)
DOCU_LONG_DESC		CHAR(240)
DOCU_SHORT_DESC		CHAR(80)
DOCU_CREATION_DATE		DATE

TABLE: DOCUMENT_VERSIONS

Name	Null?	Type
DOCU_IDENTIFIER	NOT NULL	NUMBER(5)
RELEASE_NUM	NOT NULL	NUMBER(3)
REVISION_NUM	NOT NULL	NUMBER(3)
VARIANT_NUM	NOT NULL	NUMBER(3)
STATUS	NOT NULL	CHAR(30)
CHECKED_OUT	NOT NULL	CHAR(1)
CREATION_DATE		DATE
VARIANT_TYPE		CHAR(30)

```
TEXT_FORMAT                              CHAR(30)
COMMENTS                                 CHAR(240)
LAST_MODI_DATE                           DATE
SOURCE_FILE_ID                 NOT NULL  NUMBER(5)
SOURCE_FILE_MACH_ID            NOT NULL  NUMBER(5)


TABLE:  DOCUMENT_VER_APPLVER

Name                           Null?     Type
------------------------------ --------  ---------
DOCU_IDENTIFIER                NOT NULL  NUMBER(5)
DOCU_RELEASE_NUM               NOT NULL  NUMBER(3)
DOCU_REVISION_NUM              NOT NULL  NUMBER(3)
DOCU_VARIANT_NUM               NOT NULL  NUMBER(3)
APPL_IDENTIFIER                NOT NULL  NUMBER(5)
APPL_RELEASE_NUM               NOT NULL  NUMBER(3)
APPL_REVISION_NUM              NOT NULL  NUMBER(3)
APPL_VARIANT_NUM               NOT NULL  NUMBER(3)
IS_DYNAMIC                               CHAR(1)


TABLE:  DOC_VERSION_HISTORY

Name                           Null?     Type
------------------------------ --------  ---------
DOC_IDENTIFIER                 NOT NULL  NUMBER(5)
DOC_PRED_RELEASE_NUM           NOT NULL  NUMBER(3)
DOC_PRED_REVISION_NUM          NOT NULL  NUMBER(3)
DOC_PRED_VARIANT_NUM           NOT NULL  NUMBER(3)
DOC_SUCC_RELEASE_NUM           NOT NULL  NUMBER(3)
DOC_SUCC_REVISION_NUM          NOT NULL  NUMBER(3)
DOC_SUCC_VARIANT_NUM           NOT NULL  NUMBER(3)


TABLE:  FILES

Name                           Null?     Type
------------------------------ --------  ---------
FILE_MACHINE_ID                NOT NULL  NUMBER(5)
FILE_NAME                      NOT NULL  CHAR(80)
FILE_IDENTIFIER                NOT NULL  NUMBER(6)
FILE_INFO_RES_NAME             NOT NULL  CHAR(20)
FILE_OWNER_IDENTIFIER          NOT NULL  NUMBER(10)
FILE_APPL_NAME                           CHAR(80)
FILE_SOURCE_HOSTNAME                     CHAR(20)
FILE_OBJECT_HOSTNAME                     CHAR(20)
FILE_OBJECT_NAME                         CHAR(80)
FILE_SOURCE_NAME                         CHAR(80)
FILE_APPL_VERSION_NUM                    NUMBER(3)
FILE_OBJECT_LIB_NAME                     CHAR(80)
FILE_OBJECT_LIB_HOSTNAME                 CHAR(20)
FILE_CREATION_DATE                       DATE
FILE_SUB_TYPE                            CHAR(4)
FILE_TYPE                                CHAR(40)
FILE_EDITOR_USED                         CHAR(10)
FILE_SOURCE_LANGUAGE                     CHAR(40)
FILE_RESPONSIBLE_PROGRAMMER              CHAR(240)
FILE_COMPILER_OPTIONS                    CHAR(30)
FILE_COMPILER                            CHAR(20)
FILE_LIBRARY_PROGRAMMER                  CHAR(240)
FILE_DESC                                CHAR(50)
INCLUDE_FILE_LANGUAGE                    CHAR(40)
FILE_SYSTEM_ID                 NOT NULL  NUMBER(10)
FILE_LAST_MODI_DATE                      DATE
```

TABLE: FILE_USER

Name	Null?	Type
FILE_MACHINE_ID	NOT NULL	NUMBER(5)
FILE_IDENTIFIER	NOT NULL	NUMBER(6)
USER_CAN_ACCESS	NOT NULL	NUMBER(10)

TABLE: INCLUDE_INCLUDE

Name	Null?	Type
FILE_MACHINE_ID1	NOT NULL	NUMBER(5)
FILE_ID1	NOT NULL	NUMBER(6)
FILE_MACHINE_ID2	NOT NULL	NUMBER(5)
FILE_ID2	NOT NULL	NUMBER(6)

TABLE: MACHINES

Name	Null?	Type
MACH_IDENTIFIER	NOT NULL	NUMBER(5)
MACH_NAME	NOT NULL	CHAR(20)
MACH_NETWORK_IDENTIFIER		NUMBER(3)
MACH_COST		NUMBER(6,1)
MACH_PURCHASE_DATE		DATE
MACH_VENDOR		CHAR(30)
MACH_SPEED		NUMBER(3)
MACH_MAINTAINED_BY		NUMBER(10)
MACH_LAST_CHECK_DATE		DATE

TABLE: MODULES

Name	Null?	Type
MODU_IDENTIFIER	NOT NULL	NUMBER(5)
MODU_NAME	NOT NULL	CHAR(20)
MODU_LONG_DESC		CHAR(240)
MODU_SHORT_DESC		CHAR(50)

TABLE: MODULE_VERSIONS

Name	Null?	Type
MODULE_ID	NOT NULL	NUMBER(5)
RELEASE_NUMBER	NOT NULL	NUMBER(3)
REVISION_NUMBER	NOT NULL	NUMBER(3)
VARIANT_NUMBER	NOT NULL	NUMBER(3)
COMPILER_IDENTIFIER	NOT NULL	NUMBER(4)
SOURCE_FILE_MACH_ID	NOT NULL	NUMBER(5)
SOURCE_FILE_ID	NOT NULL	NUMBER(6)
CHECKED_OUT	NOT NULL	CHAR(1)
STATUS	NOT NULL	CHAR(30)
DESCRIPTION		CHAR(50)
CREATION_DATE		DATE
SHARED_DEDICATED		CHAR(10)
OUTPUT_DESC		CHAR(240)
DEFINITION_COMPONENT		NUMBER(1)
INPUT_DESC		CHAR(240)
LAST_MODI_DATE		DATE
VARIANT_TYPE		CHAR(30)

TABLE: MODULE_VERSION_HISTORY

Name	Null?	Type
MODU_IDENTIFIER	NOT NULL	NUMBER(5)
MODU_PRED_RELEASE_NUM	NOT NULL	NUMBER(3)
MODU_PRED_REVISION_NUM	NOT NULL	NUMBER(3)
MODU_PRED_VARIANT_NUM	NOT NULL	NUMBER(3)
MODU_SUCC_RELEASE_NUM	NOT NULL	NUMBER(3)
MODU_SUCC_REVISION_NUM	NOT NULL	NUMBER(3)
MODU_SUCC_VARIANT_NUM	NOT NULL	NUMBER(3)

TABLE: MODULE_VER_APPLVER

Name	Null?	Type
MODU_ID	NOT NULL	NUMBER(5)
MODU_RELEASE_NUMBER	NOT NULL	NUMBER(3)
MODU_REVISION_NUMBER	NOT NULL	NUMBER(3)
MODU_VARIANT_NUMBER	NOT NULL	NUMBER(3)
APPL_ID	NOT NULL	NUMBER(5)
APPL_RELEASE_NUM	NOT NULL	NUMBER(3)
APPL_REVISION_NUM	NOT NULL	NUMBER(3)
APPL_VARIANT_NUM	NOT NULL	NUMBER(3)
IS_DYNAMIC		CHAR(1)

TABLE: OBJECT_EXECUTABLE

Name	Null?	Type
OBJECT_FILE_MACH_ID1	NOT NULL	NUMBER(5)
OBJECT_FILE_ID1	NOT NULL	NUMBER(6)
EXEC_FILE_MACH_ID2	NOT NULL	NUMBER(5)
EXEC_FILE_ID2	NOT NULL	NUMBER(6)

TABLE: OBJECT_LIB_EXECUTABLE

Name	Null?	Type
OBJECT_LIB_FILE_MACH_ID1	NOT NULL	NUMBER(5)
OBJECT_LIB_FILE_ID1	NOT NULL	NUMBER(6)
EXEC_FILE_MACH_ID2	NOT NULL	NUMBER(5)
EXEC_FILE_ID2	NOT NULL	NUMBER(6)

TABLE: SOURCE_INCLUDE

Name	Null?	Type
SOURCE_FILE_MACH_ID1	NOT NULL	NUMBER(5)
SOURCE_FILE_ID1	NOT NULL	NUMBER(6)
INCL_FILE_MACH_ID2	NOT NULL	NUMBER(5)
INCL_FILE_ID2	NOT NULL	NUMBER(6)

TABLE: SYSTEMS

Name	Null?	Type
SYST_IDENTIFIER	NOT NULL	NUMBER(10)
SYST_NAME	NOT NULL	CHAR(20)
SYST_COST		NUMBER(6,1)
SYST_INSTALLATION_DATE		DATE
SYST_WHERE_AVAILABLE		CHAR(30)

```
SYST_VENDOR                              CHAR(30)
SYST_PURCHASE_DATE                       DATE
SYST_NUMBER_COPIES                       NUMBER(3)


TABLE:  TABLES

Name                             Null?    Type
-------------------------------- -------- ---------
DB_OBJ_IDENTIFIER                 NOT NULL NUMBER(5)
TABL_NAME                         NOT NULL CHAR(30)
TABL_INITIAL_TRAN                 NOT NULL NUMBER
TABL_TABLESPACE_IDENTIFIER        NOT NULL NUMBER(5)
TABL_CLUSTER_COLUMNS              NOT NULL NUMBER
TABL_COLUMNS                      NOT NULL NUMBER
TABL_MAXIMUM_TRAN                          NUMBER


TABLE:  USERS

Name                             Null?    Type
-------------------------------- -------- ---------
USER_IDENTIFIER                  NOT NULL NUMBER(10)
USER_CREATION_DATE               NOT NULL DATE
USER_LOGIN_NAME                  NOT NULL CHAR(10)
USER_NAME                        NOT NULL CHAR(20)
USER_PASSWORD                             CHAR(10)
USER_SUB_TYPE                             CHAR(4)
USER_TYPE                                 CHAR(4)
SUB_TYPE                                  CHAR(4)


TABLE:  USER_APPLVER

Name                             Null?    Type
-------------------------------- -------- ---------
USER_CAN_RUN                     NOT NULL NUMBER(10)
APPL_ID                          NOT NULL NUMBER(5)
APPL_RELEASE_NUM                 NOT NULL NUMBER(3)
APPL_REVISION_NUM                NOT NULL NUMBER(3)
APPL_VARIANT_NUM                 NOT NULL NUMBER(3)


TABLE:  VIEWS

Name                             Null?    Type
-------------------------------- -------- ---------
DB_OBJ_IDENTIFIER                NOT NULL NUMBER(5)
VIEW_NAME                        NOT NULL CHAR(30)
VIEW_NUMBER_COLUMNS              NOT NULL NUMBER(3)
VIEW_TEXT_LENGTH                          NUMBER(3)
VIEW_TEXT                                 CHAR(240)
```

Appendix D

Usage of Prototype Services

RADS (Repository-based Application Development Services) is the collection of application development services developed for use with the WSRC prototype repository [15]. The primary purpose of RADS is for the management of application configurations and their constituent modules and documents. This encompasses the management of changes, versions, configuration relationships, and projects. Impact analysis functionality is also provided.

The user interface for the RADS system is menu based. Therefore, it is natural to describe the features of the system by presenting each of the menu options and suboptions in turn. This section is divided into subsections based upon the selections in the main menu: version management, project management, configurations, project management, and report generation. Through the analysis of these, all of the features of this system will be uncovered and explored.

D.1.1 Project Management

In a software development environment, a project manager is usually assigned the task of deciding who handles each part of a task within a proposed software development project.

This person will be responsible for setting up the scope of the project as well as deciding the breakdown of tasks. In turn, each task will be assigned to a developer or team of developers to work on. This option within the RADS system will be of greatest use to the project manager. Herein, a manager will define and maintain all tasks within a development project as well as the resources assigned to them. Further, this option provides the primary route of entering modules, documents, and applications into the repository.

Initiate a New Application

This option allows a new application to be initiated into the RADS system. In creating a new application, several pieces of information must be supplied. Required information includes application name, application group name, process model name, process model identifier, and an initial version number. Optional information includes input and output descriptions, short and long application descriptions, acceptance level status, linker name, linker option string, application type, executable file name, and machine name. An indicator of whether the application is checked-out to a user or not as well as the application creation and last modification dates are automatically tracked.

Define/Remove Constituent Code Modules

The second option allows for the managing of code and definitional modules associated with an existing application. It is in this area that a user would define all base source code and definitional modules that will constitute an application program. There are four suboptions offered: select an existing module, define a new module, view a list of currently associated modules, and remove a module from a configuration. Note that only one version of a module may be associated with a given application version at a time.

A module may already exist within the system. If so, a simple linkage can be established between it and a particular version of an application configuration. However, if it does not exist and the user wishes to include it in the configuration, it must be created within the RADS system. First, the user must supply basic information about the new module including a module name, initial version number, source file name, machine name, acceptance level status, file owner, and an indicator of whether the module is a code or definitional module. In order to assign an owner to a module file, the owner must be registered in the repository as a valid system user. Optional information includes input description, output description, compiler, compiler option string, and a general textual description. An indicator of whether the module is checked-out to a user or not as well as the module creation and last modification dates are automatically tracked.

Associated with any module to application relationship is an indicator of how the module should be bound to the application. A module may be either statically or dynamically bound to an application configuration. Refer to Section 3.2.1 for further details on bind indicators.

The fourth suboption provides a means of removing a particular module from an application configuration. This should also be used to alter the configuration bind indicator or to change the version of a module participating in a configuration.

Added for convenience, the third option allows the user to view all modules currently included in an application configuration. This report includes module name, version number, assigned developer, and the configuration bind type.

Define/Remove Constituent Documents

This option is nearly identical to the previous option. The same suboptions exist: select an existing document, define a new document, view associated documents, and remove a

document from the configuration.

Documents can either already exist within the RADS system or be defined using the option to define a new document. At most one version of a particular document may be associated with a given application configuration version. The link between a document version and an application version can either be static or dynamic.

Metadata similar to that maintained for modules as explained above is stored for documents as well. Required information includes: document name, version number, acceptance level status, source file name, source file host machine name, and file owner. To be able to assign an owner to a document file, the owner must be registered in the repository as a valid system user. Optional information includes textual comments and descriptions of the document. An indicator of whether the document is checked-out to a user or not as well as the document creation and last modification dates are automatically tracked.

The third option provides the user with a report of all document versions currently bound to the chosen application version. The fourth option allows the user to delete an association between a document version and an application version.

Define/Remove Associated Object Library Files

This option is similar to that of associating modules and documents to a an application version with one major difference: object library files are not strictly managed within the RADS system. That is, these files are not explicitly assigned a version number. The configuration association is based upon a simple file to application version relationship. Four options exist within this menu: select an existing object library file, define a new object library file, view list of currently associated object library files, and remove an object library file from a configuration.

When defining a new object library file in the system, the user must provide the object library file name (with file extension "a"), source file name, host machine name, and login name of the owner of the file. An optional textual file description is allowed. When selecting an existing object library file for containment within a configuration, only the object library file name and host machine name are required.

The third option allows a user to view all object library files currently associated with the chosen application version. Note that there is no issue of static versus dynamic configuration binding since object library files are not versionable (unless their corresponding source code is explicitly defined as a module). The last option allows a user to remove the association between an object library file and an application version.

Define/Remove Additional Team Members

For a software application project, a specific group of people will be responsible for all development and maintenance of the code. Moreover, security clearance for altering the components of an application should be granted exclusively to those people who are members of one of these teams assigned to work on the project. This RADS option allows a project manager to maintain the relationships between employees and application software projects. Each person involved can be assigned to one or both of two roles: that of a developer or of a maintainer. The user can either assign a person to a role for an project or remove an existing assignment. This resource assignment is the basis for security within the RADS system.

D.1.2 Change Management

Considering that more than half of a programmer's time is devoted to the maintenance of existing source code, it is clear that change management crucial service for any application development organization. Included in this RADS option are a check-in/check-out facility, a change request subsystem, and an impact analysis mechanism. To facilitate the managing and tracking of changes made to objects managed by RADS, a basic check-in/check-out model is implemented. In place of more traditional paper-based change request systems, RADS provides an automated means for issuing and approving change requests and bug fixes to managed objects. The impact analysis feature enables the effects of a change to either a database object or a software object managed by the RADS system to be assessed prior to implementing the change. All of these features will be detailed in the following subsections.

For purposes of clarity, it is necessary to define the domain of values allowed for certain change management related metadata. Associated with each module, document, and application version is a status label that indicates its level of testing or acceptance. Modules and applications can have a status of *working*, *unit tested*, *system tested*, or *approved*. Documents can have a status of *draft*, *under review*, or *approved*. Note that although a status must be attached to all managed software element versions, it is not required that the ones listed above are strictly adhered to. However, if a non-traditional status labeling scheme if opted for, the automatic tracking of application version acceptance status cannot be done.

Check Out Menu

There are two types of check outs available within RADS: read-only and editable. In both cases, the user must be eligible to check out a particular element version. That is, the user

must be registered in the repository as a valid system user and be a member of either the development or maintenance team of an application that contains that particular element. In order to check out an element version in the editable mode, the item cannot already be checked out and it must be a terminal version in the main version line or upon a branch. A user can check out the version in the read-only mode in any case, however. If a user desires to check out either a non-terminal or already checked out element version, the version management menu option must be consulted to create a variant branch upon which to make revisions.

There are times when a user would simply want to look at the source file of a particular element version. In this case, the user would use a read-only check out. In a read-only check out, the file is copied to the user's current directory: no metadata is maintained about who has checked out the file. This type of check out is basically a means to recreate a file: the file is not truly checked out and therefore, can not be checked back in. All of the managed object types (documents, modules, and application configurations) can be checked out on a read-only basis. When checking out a complete application, the user has the option to check out either all contained module versions or all contained document versions.

In most cases, however, a user would want to check out an element version to make changes upon. Only individual documents and modules can be checked out for making revisions. Upon checking out an element version in the editable mode, the corresponding source file is copied into the user's current working directory and the file permissions are set to allow for writing. The user is free to make any changes to the file until the file is checked back into the RADS system library.

Check In Menu

Check-ins are only allowed for modules and documents managed by the RADS system. There are two types of check-ins: reserved and unreserved. Reserved check-ins are recorded within the repository and the source file changes are stored in the library. On the other hand, software items checked in using the unreserved mode are labeled as being checked in but all changes to the source are disregarded. This mode should be used when a developer basically wants to un-checkout a particular element version without any of the changes being recorded.

There are several restrictions that must be adhered to in the reserved check in process. First, the source file for the element version being checked in must reside in the user's current working directory. Second, the user must be eligible to check in the item; that is, the user must either be the owner of the file or a member of a development or maintenance team associated with any application configuration that contains the element. Last, a user cannot check in an element that is not checked out.

Depending upon the acceptance status level of the element version being checked in, there are two ways that version number incrementing can occur. If an element version has the lowest status level (*working* for modules or *draft* for documents) the user can opt to either increment the version number of the element in turn forcing a new copy to be initiated into the library or to not increment the version number and in turn overwrite the code of the previously existing element version. However, if the status level is not at the lowest level, the revision number is automatically incremented upon check in with no exceptions.

The status of an element can be updated upon check in. There is a facility that allows the user to automatically upgrade or downgrade the status based upon the standard acceptance

status levels that RADS offers. Alternatively, the user can simply change the status to be any arbitrary string. If the status of the element is upgraded or downgraded, a check is done to see if the status of any application configuration version that the element participates in must be updated as a result. Note that the status of an application version can only be as high as the lowest status attained by all of its contained module versions.

Modules and documents are allowed to participate in both statically and dynamically configured links to their parent application configurations. Any time the version number of an element is incremented as a result of a reserved check in, any dynamically configured links that the item participates in must be automatically updated. More specifically, if the item that is being checked in participates in any dynamic configurations to an application version, each such linkage is updated to include the new element version.

For code and definitional modules, RADS performs an additional analysis. On check-in, the source file of each module is scanned for header files included using the *#include* directive. For each such include file identified, if it is represented within the repository then a metadata linkage is created between the module version and the include file. Through this process, useful application build information about source file dependencies is attained.

If the version number of an include file (defined as a definitional module) is incremented, RADS updates all source file to include file relationships. For each active application configuration version to which the definitional module is dynamically bound, a list of all module versions that participate in the configuration is generated. If any of these module versions were associated with the previous definitional module version, the linkage between the code module version and the definitional module version is updated to include the new definitional module version.

Change Request Menu

The change request system provides an automated means of tracking change requests throughout their life cycle. A particular change request is associated with a specific module version. It has an associated status indicating its current stage in its life cycle. A request can have a status of *under review*, *approved*, *implemented*, or *rejected*. In addition, much metadata is maintained for each change request: change request number, the specific module version the request was issued for, the description and reasons for the change to be made, the person who originated of the request, the priority of the change, the person who approved the request, the approval date, approval comments, the person who implemented the changes, implementation date, implementation comments, and the module version in which the change was implemented.

There are several options available in the change request system. First, a user can initiate a change request. This option allows the user to create a new change request for a specific module version. Information about the problem and the changes to solve it must be entered. As a result of creating a new change request, it will be initiated into the system with the status of *under review*. One of the primary functions needed in a change request system is a simple way to check the status of existing change requests in the system. The second option allows a user to check a change request status using one of two access routes: by change request number (provides a detailed description of one particular change request) or by module (provides a more general description of all change requests related to that module). The third option allows a user to change the status of a change request. Typically the user who either approves, implements, or rejects a particular change request would use this option to change the status of the request and enter detailed information

regarding the new status. The fourth option provides the user with a quick listing of all outstanding change requests in the system. In effect, this lists basic information about all change requests whose status is either *under review* or *approved*; that is, change requests that are not implemented and therefore are either complete or rejected. Note that for proper functioning of the status based system, the predefined status levels should be conformed to.

To better illustrate how this feature could be used, consider the case of a small software development firm working on a new personal budget management application. This firm is divided into four primary groups: new code developers, code maintainers, code testers, and managers. While performing routine testing of a particular code module, a tester finds that a mathematical computation error is occurring. Upon finding this error, he initiates a change request in the RADS system outlining the details of the error. On a regular basis, a manager will use the change request reporting feature to obtain a listing of outstanding change requests. At this time she could isolate the new change requests that have a status of *under review* and review them to see if given the company's time constraints, resources, and the complexity of the problem the request should be implemented or rejected. She would then set the status appropriately. A member of the maintenance team periodically checks the list of outstanding change requests to see which ones have been approved and are ready for implementation. Upon seeing that a request has been approved, he will proceed to implement the change. Once the implementation is complete, he will use the change request system to change the status of the change request to *implemented* and to enter the details and limitations associated with the implementation. Therefore, the company has an efficient means for tracking problems and their corresponding fixes.

Impact Analysis Report/Initiation

In a highly diverse and integrated software development environment, it is crucial to be able to gauge the effect that a change to a particular software object may have an other existing objects. This RADS option provides the user with two main functions: generating an impact analysis report and notifying of the affected user groups of a change.

The RADS system supports impact analysis where the object to be changed is a database object (table, column, index, trigger, sequence, synonym), a definitional or code module, a document, or a general file, any of which must be managed by the repository. Due to the design of the repository meta model upon which the RADS system has been implemented, there is one major limitation: a change to any of the above items can only be traced to the effect it will incur to a particular software application configuration. Upon isolating the software item to be changed, an impact analysis report is generated and displayed to the user. Included in this report is a listing of all of all terminal application versions affected by this change.

At this point, the user can choose to either initiate the change notification process or exit the option. If the change notification process is initiated, the user will be prompted for several pieces of information which will be included in the notification report. To aid the affected user community in understanding and appropriately reacting to the change, details about and reasons for the change are requested from the initiating user. In addition, if the change is associated with a specific change request in the system, it is noted. All of this information plus the date and originator of the change is consolidated into a notification report. This report is distributed through the *UNIX* electronic mail system to all members of the affected user community. This community consists of all members of the development

and maintenance groups of the applications affected by the change. A sample notification report is presented in Figure ??.

D.1.3 Version Management

The options available in the version menu all deal with performing some alteration of the version history of either a module, a document, or an application. Included options are these: link the version history of two element versions, increment the release number of an element, create a variant branch, and perform a merge of two elements.

Link Version History of Two Versions

The standard version history created by a sequence of check-out/check-in operations results in a linear progression down either a main branch, a variant branch, or both. There are instances, however, when a developer may opt to note that a particular version not only resulted from its immediate predecessor (resulting from a check-out/check-in operation) but also from some other version. This option allows a history linkage to be created between two specified versions of either a module or a document.

The most common use of this option would be in indicating that a merge was performed between two element instances to create a new version. Consider the case where two developers wish to modify the *parser* module version *2.5.0*. The first developer successfully checks out the code for making revisions. Since the first developer has the current version checked out, the second developer will have to create a variant branch extending from this version to make revisions upon. After both have made the necessary alterations to their files and checked the changes back in, a merge will need to be performed using the merge option. The merge option does not perform the version history linkage of the two merged

versions for the developer: it is required that the linkage be explicitly set using this option. More details on creating variant branches and merging element versions will be given later in this section.

Increment Release Number of an Element

At some critical time in the life cycle of an element, a developer may wish to note a significant change in the functionality or content of an element version by incrementing the release number of that element. For example, a developer may wish to increment the release number at each acceptance status level change that the element undergoes, or perhaps at times when a significant functional change has been enacted within a code module. In these cases, the developer may use this option to increment the release number of either a module or document element.

When using a basic check-out/check-in sequence of making changes to a module or document element, only the revision and variant numbers are incremented: the release number remains constant. Note that when a particular application is released, none of the release numbers of the constituent elements are incremented; only the release number of the application version itself is incremented. Incrementing the release number of the involved element versions would not be feasible since an element may not be exclusive to a given configuration. Therefore, this is the only route to increment the release number of an element version.

When the release number of a basic element is incremented, the revision and variant numbers are both reset to 0. A linkage is then made between what was the terminal element version on the main branch and this new released version. The result of increasing the release number of a particular element will be to freeze the current version of the element in the

main version branch from further alteration as well as to reconfigure any dynamic links the particular element may have to existing application configurations. For example, consider that the *stack* module with version *5.3.0* is to be released. This module participates in a dynamic linkage to two applications, *drgcompiler* and *mephisto*. Upon incrementing the version number of the element to *6.0.0* the application configurations of the current version of *drgcompiler* and *mephisto* will be updated to include version *6.0.0* instead of version *5.3.0* of the *stack* module. There is no configuration change required when dealing with static configuration linkages.

Create a Variant Branch

One of the primary functions expected of any version management system is that of support for concurrent or parallel development. A related situation would be that of a developer needing to make revisions, such as a bug fix, to a version of an element that is frozen. These events must occur in a very controlled environment to ensure that all changes made by each developer are maintained in their entirety. The parallel revision scheme used by the RADS system revolves around the explicit creation of a temporary variant branch upon which to make revisions.

Consider the following situation: one developer needs to make changes to the current version of a particular document. Upon attempting to check-out this element version for revision, she finds that the version is already checked out to another user. At this point, she has two choices. First, she could wait until the other developer completes his revisions and checks the document back in. A more desirable choice, however, would be for her to instead of waiting, make revisions upon the same initial document version that the other developer started with, then upon each of them checking in their revisions, perform a merge

to accumulate all of their changes into a new document version. To facilitate the second developer's desire to perform concurrent revisions, RADS allows for the creation of variant branches originating from a particular version of an element.

Since RADS uses a purely numeric, three level versioning scheme, only one variant branch can originate from a particular element version source. For example, for a particular document version in the main branch with version number *2.5.0*, a variant branch could be created with the first instance on this branch having version number *2.5.1*. Upon each revision of the element along this variant branch, the variant number would be incremented (e.g. *2.5.1*,*2.5.2*, *2.5.3*,...). In the case of variant branches, the revision number indicates the origin of the branch and does not change upon each variant branch revision. Upon creating a variant branch, the new element version is not automatically checked out for revision. However, it is available for check out using the change management option.

Perform a Merge of Two Elements

Since one of the primary features of a version management system is support for concurrent development, the reverse functionality must be provided that bundles up all of the changes performed upon a single item into one new element instance. Consider the situation where two developers have made revisions to a particular code module version. By the structure of the concurrent revision subsystem within RADS, it is required that one developer makes a revision to an element in the main branch whereas the other developer makes a revision to an element in a variant branch. Upon each developer finalizing and checking in their revisions, the two versions should be merged into one version located in the main branch.

RADS supports a basic two text file merge facility. When considering performing a merge of two text files, RADS requires three files to be identified: the filename of the

contributor in the main branch, the filename of the contributor from the variant branch, and the filename of their closest common ancestor. Note that the files under consideration should be of the same module or document and must be checked out.

Following the traditional merge algorithm, each of these three files are compared line by line concurrently. If the situation where all three files differ at a particular point is not encountered, a consolidated merge file will be created. However, if a collision does occur, the user will be notified and no merged file will be generated. This situation requires that either a manual merge is done by the user or perhaps done by a tool outside of the RADS system. If the merge file is created, it will be placed into the user's current directory into a file whose name is *merged.file*. It is wise to closely analyze the generated merge file for accuracy to determine if it is as expected. Note that the generated file is not automatically checked in. This and the version history linkage must be done separately by the user.

The following example will outline entire merging process. Consider that one developer has checked out version *6.4.0* of the *parser* module to make revisions upon (refer to Figure D.1a). A second developer wishes to make revisions to the same module version. The following steps should be followed to allow for concurrent development terminated with a merge.

1. The second developer must create a variant branch originating from *parser* version *6.4.0*.

2. The second developer should then check this element version out for revision. Refer to Figure D.1b.

3. Both developers should perform the necessary revisions and check in the files.

4. Both files to be merged should be checked out at this point. An editable check-out

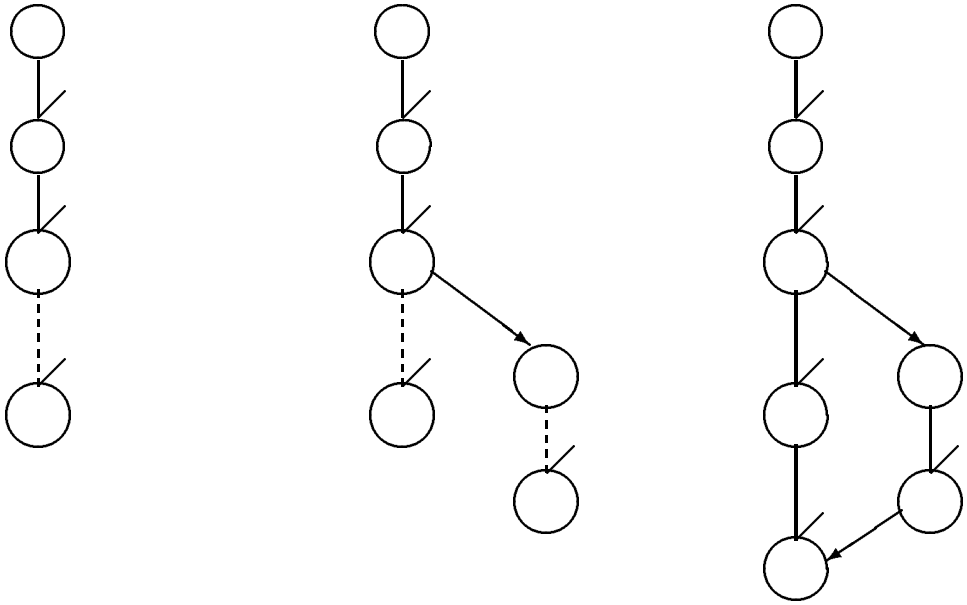

Figure D.1: RADS Merging Process

should be performed for the main branch contributor whereas a read-only check-out should be performed for both the variant branch contributor and their latest common ancestor.

5. Execute the merge option giving first the filename of the file from the main branch (code of first developer), the filename of the file from the variant branch, then the filename of their latest common ancestor.

6. If the merge is nonconflicting, the merged file will be created.

7. Each developer should verify that the merged code contains each of their changes.

8. If the merged file is acceptable, it should be checked in as the next revision in the main version branch. Refer to Figure D.1c.

9. A link from the latest version in the variant branch to the latest version in the main branch for the particular module or document should be made.

This completes the merging process. The primary reason for not automating all of these steps within the merge process is to avoid problems that could result from forcing incorrect merged files to be permanently saved within the system.

D.1.4 Configurations

The focus of this option is on services involving entire application configurations. Options include an automated configuration build option that generates a *makefile* for an application version, a configuration release option that provides the user with an automated means of issuing a release of an application configuration, an option to determine and extract the relationships between an application configuration version and any referenced database objects, and an option to display the status and all components within a configuration.

Automated Configuration Build

Determining all components that compose a particular application configuration version and all of their interdependencies is not typically a trivial task for the developer to perform. To aid in this endeavor, the RADS system provides the user with a facility for identifying all component versions of a specific application configuration version and subsequently generating a UNIX *makefile* based on this information. Note that the *makefile* is generated but not executed by RADS: the user is responsible for executing the build.

First the user must select a specific application configuration version to create the *makefile* for. Before the *makefile* is created, one of two conditions must be satisfied: either all components of the application version must be checked in or the entire configuration must

be checked out. This restriction requires that the configuration be in a stable state at the time of *makefile* creation; that is, no development work is currently being done to the components of the configuration and therefore, the version selection is done correctly.

Next, the user must specify the full directory path to be used to locate the source files, object files, and executable file. For each of these, the path can either be the standard RADS library path as the default or some other path that the user specifies. It is suggested that the default path be used only for the object and executable files and that a user defined path be used for the source files. Note that since delta storage is used to store non-terminal source files, it is necessary to check out all components of the configuration at one time to guarantee that the build is being done to source files, not to their corresponding delta files. If the default object file path is selected, maximal reuse of already compiled source code is attained. The executable path determines where the executable program file will be placed once created.

At this time the *makefile* is generated. The user can either select the default *makefile* name (named *makefile* in the user's current directory) or specify some other path and file name. The repository tracked relationships between application configurations and object files, executable files, object library files, source code files, and header files are used to define the compilation and linking dependencies for the configuration. Executable files are dependent upon object and object library files. Object and object library files are dependent upon their corresponding source files and all recursively associated include files. With this relationship information as well as the linker and linker option data for the application and the compiler and compiler option data for each component, the *makefile* is created. A sample generated makefile is presented in Figure **??**.

Check Configuration Status

During all phases of the life cycle of an application program, it is necessary to be able to obtain a status of all the components involved in the configuration of the application. Included in this report for each document or module version in the configuration version is an indicator of whether the item is checked out or not, the owner of the file, and the acceptance level attained. Summary information includes the number of documents and modules that are checked out.

One of the primary uses of this feature is to determine whether a particular version of an application is ready for building or perhaps, for releasing. A sample status report is presented in Figure **??**.

Update Application-Database Object Association

One of the primary advantages of repository baseds configuration management systems over traditional configuration management systems is the ability to represent and track the relationships between different types of software items. The usefulness of this functionality is clear in specialized software development arenas. For instance, of special concern within a database application development arena is the impact a change to a particular database object may have not only with respect to other database objects but also with respect to all source code (and in turn, applications) that reference the database objects.

The repository meta model underlying the RADS system supports database object to application version relationships on the level of any database object (table, view, column, index, sequence, synonym) to a particular application configuration version. At this time, RADS only supports the database table or view to application version relationship.

This option will allow the user to determine and create application version to database

table and view relationships within the repository. To obtain this relationship information, each code module version associated with the selected application version is parsed for standard SQL constructs. From these blocks of code, the involved table and view names are extracted. If the database table or view is managed by the repository, the relationship is recorded. Note that only relationships to tables and views maintained by the repository can be tracked; no error will occur if the table or view is not found in the repository, however.

Note that when a code module is checked in, it is not automatically parsed to obtain these database object relationships. This analysis must be requested explicitly via this option. The primary application of this feature is in the arena of impact analysis.

Issue Release

At certain times in the life cycle of an application program, notable milestones are reached. For instance, a major change in functionality is implemented or perhaps, a significant collection of bug fixes is completed. At these critical points in the development and maintenance cycle, it is beneficial to note these changes not only within the documentation but also within the versioning scheme.

Prior to releasing an application, two initial conditions must be met: all element versions in the application configuration must be checked in and each element version as well as the entire application must have achieved a high level of acceptance and testing.

The process of issuing a release is composed of several tasks. Within the RADS system, there are five major steps that occur when a particular application version is released:

- A new release subdirectory is created in the REL subdirectory which is named by appending the version number to the application name. (For example, for an application named *hyperpascal* with release version *4.0.0*, the directory would be named

hyperpascal.4.0.0.

- All documentation files in the configuration are copied into the new release subdirectory.
- The executable code is copied from the BIN directory into the new release subdirectory.
- A short report is generated that indicates all elements participating in the configuration along with their version numbers. This report is placed into the new release subdirectory.
- The version number will be changed: the new release number will be one more than the current release number, the revision number will be 0, and the variant number will be 0. As a result, the current application version will be frozen. The new application version will be reconfigured so that it is initially identical to the state of the preceding version.

There are several constraints, however, that the user must adhere to prior to issuing a release:

- All element versions in the application configuration to be released must be checked in.
- All element versions in the application configuration to be released must have reached the highest acceptance status level.
- The executable code for the application must be built and reside in the RADS library BIN subdirectory.

D.1.5 Report Generation

One of the primary characteristics that separates repository based configuration management systems from traditional configuration management systems is the amount and detail of metadata stored and tracked for all elements under the control of the repository. Therefore, some of the most pertinent and useful information can be gathered from reports on the the status of the elements and configurations within the system. Four main reports are offered within this option: a change history report, a version history report, a object catalog report, and a complete application configuration report. Version history, change history, and catalog reports are available for modules, documents, and applications managed by the repository. For each report type, the report is both displayed page by page to the screen as well as placed into a text file whose name is specified by the user.

Change History Report

The focus of the change history report is to fully describe all existing versions of a particular element (module, document, or application) and the code changes enacted that resulted in each of these versions. Upon the checking in of any controlled element, the user is prompted to enter a description of the change. This information is tracked for each version of an element and is displayed for each within this report. Other information provided includes the name of the user who made the change, the date of the modification, and any change request associated with the change. A sample change history report for a module is presented in Figure **??**.

Version History Report

This report is similar to the change history report in that all existing versions of an element are detailed. However, the focus here is not the information regarding what changes resulted in each new element version but in the relationships between them.

For each element instance, all immediate successors in the version history of the element are listed. This complete listing of relationships will completely describe the version history tree of the element in question. A sample version history report is presented in Figure **??** and its corresponding version history tree in Figure **??**.

Object Catalog Report

This report allows the user to view basic information about all modules, documents, and applications currently managed by the RADS system. Included in the report is the object name and a short description of the object. For modules, there are two catalog reports available: the basic report and a detailed functional-level report. Included in the detailed report is the module name, description, current version number, and a listing of all functional definitions included within the file. This functional listing is generated by parsing the source file for all function definition, data structure definition, and pre-compilation directives.

The primary use of these reports is allow users to identify and perhaps benefit from reusing existing software items. Note that once this basic information is attained, further object version metadata can be retrieved using one of the other reports available. A sample object catalog report is presented in Figure **??**.

Complete Application Report

The most complete and comprehensive report that the RADS system generates is the overall application report. The information in this report includes detailed information on all code and definitional module versions and document versions participating in the selected configuration. In addition, information regarding all associated object library files is given. Maintenance and development group member listings are also provided.

For each code or definitional module associated with the particular application version, all metadata maintained for it is displayed as well as a recursive listing of all include files associated with it. The metadata is not only the basic module metadata but also any metadata associated with the corresponding source file itself. Similarly, all metadata associated with constituent document versions is displayed. The best means of illustrating the depth of the overall application version report is by example. Refer to Figure **??** for a sample complete application report.

Appendix E

Sample Reports

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
                        Change  Notification  Report
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*


Changed Object:  Module
Module Name:  hp_parser

Release:  1
Revision:  1
Variant:  0

Date of Change:  15-APR-95
Initiator of Change:  edwards


Reason For Change:  The implementation of the parser module is be
                    ing revised.  It will not affect its current
                    usage.

Details of Change:  Error checking will be implemented in a loose
                    r fashion.  No outside impact.



Affected Application Versions:
-------------------------------------------
hyperpascal                 1.1.0
```

Figure E.1: Impact Analysis Change Notification Report

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
          Status Report For Application:  hyperpascal 1.1.0
                                    Date:  15-APR-95
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*


Application Information:
========================
Description:  Hyperpascal compiler
Status:  UNIT TESTED
Last Modified:  15-APR-95
Checked Out:  N
Executable Name:  /home/grad/edwards/DEV/BIN/hyperpascal.1.1.0.
Executable Host Name:  pollux



Associated Code Modules:
========================

Module               Version    Last Modified  CO?  Status          Owner
-------------------------------------------------------------------------
hp_aux               1. 1. 0    15-APR-95      Y    UNIT TESTED     edwards
btree                1. 1. 0    15-APR-95      N    UNIT TESTED     edwards
hp_error             1. 1. 0    15-APR-95      N    UNIT TESTED     leward
hp_graph             1. 1. 0    15-APR-95      N    UNIT TESTED     edwards
hpc                  1. 1. 0    15-APR-95      N    UNIT TESTED     leward
hpc_header           1. 1. 0    15-APR-95      N    UNIT TESTED     edwards
stack                1. 1. 0    15-APR-95      N    UNIT TESTED     leward
queue                2. 0. 0    15-APR-95      N    SYSTEM TESTED   leward
hp_parser            1. 1. 0    15-APR-95      N    UNIT TESTED     edwards
scanner              1. 1. 0    15-APR-95      N    UNIT TESTED     edwards



Associated Documents:
=====================

Document             Version    Last Modified  CO?  Status          Owner
-------------------------------------------------------------------------
hp_bnf               1. 1. 0    15-APR-95      N    UNDER REVIEW    edwards
hp_usage             1. 1. 0    15-APR-95      N    UNDER REVIEW    leward


Summary Information:
====================
1 code module(s) are checked out.
All documents are checked in.
```

Figure E.2: Sample Configuration Status Sample Report

```
#
# Makefile Generated By RADS
#

# Macros Defined
LINKER=gcc
LINKER_OPT=
SRC_PATH=
OBJ_PATH=/home/grad/edwards/DEV/OBJ/
EXE_PATH=/home/grad/edwards/DEV/BIN/


#Executable

$(EXE_PATH)hyperpascal.1.1.0.: $(OBJ_PATH)hp_aux.1.1.0.o $(OBJ_PATH)hp_parser.1.1.0.o
$(OBJ_PATH)scanner.1.1.0.o $(OBJ_PATH)hp_graph.1.1.0.o $(OBJ_PATH)btree.1.1.0.o
$(OBJ_PATH)queue.2.0.0.o $(OBJ_PATH)stack.1.1.0.o $(OBJ_PATH)hpc.1.1.0.o
$(OBJ_PATH)hp_error.1.1.0.o
$(LINKER)  $(LINKER_OPT)  -o $(EXE_PATH)hyperpascal.1.1.0. $(OBJ_PATH)hp_aux.1.1.0.o
$(OBJ_PATH)hp_parser.1.1.0.o $(OBJ_PATH)scanner.1.1.0.o
$(OBJ_PATH)hp_graph.1.1.0.o $(OBJ_PATH)btree.1.1.0.o
$(OBJ_PATH)queue.2.0.0.o $(OBJ_PATH)stack.1.1.0.o
$(OBJ_PATH)hpc.1.1.0.o $(OBJ_PATH)hp_error.1.1.0.o

#Associated Object Libraries


#Associated Source Files

$(OBJ_PATH)hp_aux.1.1.0.o: $(SRC_PATH)hp_aux.1.1.0.c $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)hp_aux.1.1.0.c

$(OBJ_PATH)hp_parser.1.1.0.o: $(SRC_PATH)hp_parser.1.1.0.c $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)hp_parser.1.1.0.c

$(OBJ_PATH)scanner.1.1.0.o: $(SRC_PATH)scanner.1.1.0.c $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)scanner.1.1.0.c

$(OBJ_PATH)hp_graph.1.1.0.o: $(SRC_PATH)hp_graph.1.1.0.c $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)hp_graph.1.1.0.c

$(OBJ_PATH)btree.1.1.0.o: $(SRC_PATH)btree.1.1.0.c $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)btree.1.1.0.c

$(OBJ_PATH)queue.2.0.0.o: $(SRC_PATH)queue.2.0.0.c  $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)queue.2.0.0.c

$(OBJ_PATH)stack.1.1.0.o: $(SRC_PATH)stack.1.1.0.c $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)stack.1.1.0.c

$(OBJ_PATH)hpc.1.1.0.o: $(SRC_PATH)hpc.1.1.0.c $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)hpc.1.1.0.c

$(OBJ_PATH)hp_error.1.1.0.o: $(SRC_PATH)hp_error.1.1.0.c $(SRC_PATH)hpc_header.1.1.0.h
gcc -c  $(SRC_PATH)hp_error.1.1.0.c
```

Figure E.3: Sample Generated Makefile

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
         Change History Report For Module:  queue
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*


Version:  2.0.0      Last Modified:  15-APR-95
Status:  SYSTEM TESTED       Owner:  leward
Description:  Revised enqueue function.

-------------------------------------------------

Version:  1.5.0      Last Modified:  15-APR-95
Status:  SYSTEM TESTED       Owner:  leward
Description:  Revised enqueue function.

-------------------------------------------------

Version:  1.4.0      Last Modified:  15-APR-95
Status:  UNIT TESTED         Owner:  leward
Description:  Result of merge

-------------------------------------------------

Version:  1.3.0      Last Modified:  15-APR-95
Status:  UNIT TESTED         Owner:  leward
Description:  Revised documentation

-------------------------------------------------

Version:  1.2.2      Last Modified:  15-APR-95     (CR 1)
Status:  UNIT TESTED         Owner:  leward
Description:  Added view_queue function

-------------------------------------------------

Version:  1.2.1      Last Modified:  15-APR-95
Status:  UNIT TESTED         Owner:  leward
Description:  Added documentation

-------------------------------------------------

Version:  1.2.0      Last Modified:  15-APR-95
Status:  UNIT TESTED         Owner:  leward
Description:  Added documentation

-------------------------------------------------

Version:  1.1.0      Last Modified:  15-APR-95
Status:  UNIT TESTED         Owner:  leward
Description:  Initial code entry

-------------------------------------------------

Version:  1.0.0      Last Modified:  15-APR-95
Status:  WORKING             Owner:  leward
Description:  Specialized queue module for hyperpascal

-------------------------------------------------
```

Figure E.4: Change History Report

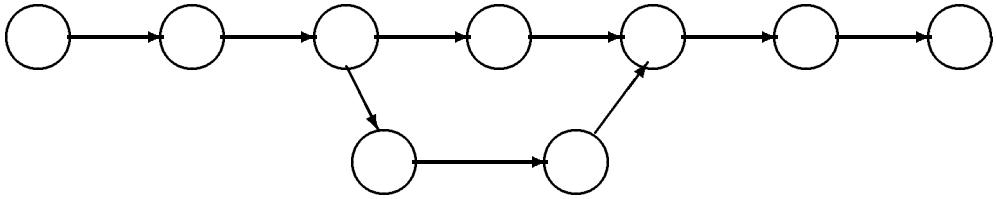

Figure E.5: Module Version History Tree

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
          Version History Report For Module:  queue
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

Object Version  ::>  Successor Versions

1.0.0 (WORKING, 15-APR-95) ::>   1.1.0

1.1.0 (UNIT TESTED, 15-APR-95) ::>   1.2.0

1.2.0 (UNIT TESTED, 15-APR-95) ::>   1.2.1,  1.3.0

1.2.1 (UNIT TESTED, 15-APR-95) ::>   1.2.2

1.2.2 (UNIT TESTED, 15-APR-95) ::>   1.4.0

1.3.0 (UNIT TESTED, 15-APR-95) ::>   1.4.0

1.4.0 (UNIT TESTED, 15-APR-95) ::>   1.5.0

1.5.0 (SYSTEM TESTED, 15-APR-95) ::>   2.0.0

2.0.0 (SYSTEM TESTED, 15-APR-95) ::>   None.
```

Figure E.6: Module Version History Report

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
                    Catalog  of  Existing  Modules
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*



Module Name  Description
================================================================

hpc                 Hyperpascal main module
*** Current Version:  1.1.0  (Code Module)


   ////////////////////////////////////////////


 include "hpc.h"

 void main ( int argc , char * argv [ ] )

 void get_pgmname ( int numargs , char * args [ ] )

   ////////////////////////////////////////////


    ---------------------------------------

queue               Specialized queue module for hyperpascal
*** Current Version:  1.1.0  (Code Module)


   ////////////////////////////////////////////


 include "hpc.h"

 int is_empty_q ( queue q )

 void create_q ( queue * q )

 int enqueue ( queue * q , int task_id , int processor )

 int dequeue ( queue * q , int * task_id , int * processor )

   ////////////////////////////////////////////
```

Figure E.7: Detailed Module Object Catalog Report (Reduced Version)

Figure E.8: Complete Application Report (Reduced Version)

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
APPLICATION  VERSION  REPORT DATE:  15-APR-95
                            USER:  edwards
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

Application Name:  hyperpascal
Release Number:  1
Revision Number:  1
Variant Number:  0
Group Name:  compiler
Short Description:  Hyperpascal compiler
Long Description:  Compiler for the hyperpascal language.  Hyper
                   pascal is a language for programming with the
                    Hypercube, a parallel MIMD machine.  This ap
                   plication essentially converts hyperpascal co
                   de into Hypercube C language which then can b
                   e compiled us
Input Description:  File written in the hyperpascal language
Output Description:  File written in Hypercube/C language
Application Type:
Creation Date:  15-APR-95
Last Modified:  15-APR-95
Status:  UNIT TESTED
Checked Out:  N
Executable Name:  /home/grad/edwards/DEV/BIN/hyperpascal.1.1.0.
Executable Host:  pollux
Linker:  gcc
Linker Options:


============================================================
   ASSOCIATED  MODULES
============================================================


Module Name:  hp_parser
Release Number:  1
Revision Number:  1
Variant Number:  0
Is Definition Component:  N
Short Description:  Hyperpascal parser module
Long Description:  Module to perform a recursive descent parse o
                   n a hyperpascal program.  Input comes from st
                   din.  Panic error recovery implemented.  Type
                    checking is also implemented.
Last Modified:  15-APR-95
Owner:  Bonnie M. Edwards
Status:  UNIT TESTED
Checked Out:  N
Dynamically Configured:  Y
Source Name:  hp_parser.1.1.0.c
Source Host:  pollux
Source File Description:  Initial code entry
Compiler:  gcc
Compiler Options:
Object Name:  hp_parser.1.1.0.o
Object Host:  pollux

Include Files:
```

```
- - - - - - - - - - - - - - - - - - - - -

Filename:  hpc_header.1.1.0.h
Hostname:  pollux
Description:  Initial definition entry
Owner:  Bonnie M. Edwards
Last Modified:

- - - - - - - - - - - - - - - - - - - - -

------------------------------------------------------------

Module Name:  scanner
Release Number:  1
Revision Number:  1
Variant Number:  0
Is Definition Component:  N
Short Description:  Hyperpascal scanner module
Long Description:  Token scanner module for the Hyperpascal lang
                   uage.  Input is taken from stdin.
Last Modified:  15-APR-95
Owner:  Bonnie M. Edwards
Status:  UNIT TESTED
Checked Out:  N
Dynamically Configured:  Y
Source Name:  scanner.1.1.0.c
Source Host:  pollux
Source File Description:  Initial code entry
Compiler:  gcc
Compiler Options:
Object Name:  scanner.1.1.0.o
Object Host:  pollux

Include Files:

- - - - - - - - - - - - - - - - - - - - -

Filename:  hpc_header.1.1.0.h
Hostname:  pollux
Description:  Initial definition entry
Owner:  Bonnie M. Edwards
Last Modified:

- - - - - - - - - - - - - - - - - - - - -

------------------------------------------------------------

Module Name:  hp_graph
Release Number:  1
Revision Number:  1
Variant Number:  0
Is Definition Component:  N
Short Description:  Computation graph routines for hyperpascal
Long Description:  This module contains the functions needed to
                   implement the conversion of the computation g
                   raph description to coded form.  Also include
                   d is the processor to task assignment algorit
                   hm and associated functions.
Last Modified:  15-APR-95
Owner:  Bonnie M. Edwards
Status:  UNIT TESTED
Checked Out:  N
Dynamically Configured:  Y
Source Name:  hp_graph.1.1.0.c
Source Host:  pollux
```

```
Source File Description:  Initial code entry
Compiler:  gcc
Compiler Options:
Object Name:  hp_graph.1.1.0.o
Object Host:  pollux

Include Files:

- - - - - - - - - - - - - - - - - - - - -

Filename:  hpc_header.1.1.0.h
Hostname:  pollux
Description:  Initial definition entry
Owner:  Bonnie M. Edwards
Last Modified:

- - - - - - - - - - - - - - - - - - - - -

------------------------------------------------------------

Module Name:  hpc_header
Release Number:  1
Revision Number:  1
Variant Number:  0
Is Definition Component:  N
Short Description:  Hyperpascal include file
Long Description:  Header file for the hyperpascal application
Last Modified:  15-APR-95
Owner:  Bonnie M. Edwards
Status:  UNIT TESTED
Checked Out:  N
Dynamically Configured:  Y
Source Name:  hpc_header.1.1.0.h
Source Host:  pollux
Source File Description:  Initial definition entry
Compiler:  gcc
Compiler Options:
Object Name:  hpc_header.1.1.0.o
Object Host:  pollux

Include Files:

- - - - - - - - - - - - - - - - - - - - -

Filename:  ctype.h
Hostname:  pollux
Description:  ctype.h C header file
Owner:  system
Last Modified:

- - - - - - - - - - - - - - - - - - - - -

Filename:  stdlib.h
Hostname:  pollux
Description:  stdlib.h C header file
Owner:  system
Last Modified:

- - - - - - - - - - - - - - - - - - - - -

Filename:  stdio.h
Hostname:  pollux
Description:  stdio.h C header file
Owner:  system
Last Modified:

- - - - - - - - - - - - - - - - - - - - -
```

```
------------------------------------------------------------

Module Name:  btree
Release Number:  1
Revision Number:  1
Variant Number:  0
Is Definition Component:  N
Short Description:  Binary tree functions
Long Description:  Binary tree functions for the hyperpascal app
                   lication program.
Last Modified:  15-APR-95
Owner:  Bonnie M. Edwards
Status:  UNIT TESTED
Checked Out:  N
Dynamically Configured:  Y
Source Name:  btree.1.1.0.c
Source Host:  pollux
Source File Description:  First code entry
Compiler:  gcc
Compiler Options:
Object Name:  btree.1.1.0.o
Object Host:  pollux

Include Files:

- - - - - - - - - - - - - - - - - - - - -

Filename:  hpc_header.1.1.0.h
Hostname:  pollux
Description:  Initial definition entry
Owner:  Bonnie M. Edwards
Last Modified:

- - - - - - - - - - - - - - - - - - - - -

------------------------------------------------------------

Module Name:  stack
Release Number:  1
Revision Number:  1
Variant Number:  0
Is Definition Component:  N
Short Description:  Stack primitive functions
Long Description:  Stack primitive functions, array implementati
                   on.
Last Modified:  15-APR-95
Owner:  M. Leanne Ward
Status:  UNIT TESTED
Checked Out:  N
Dynamically Configured:  Y
Source Name:  stack.1.1.0.c
Source Host:  pollux
Source File Description:  Initial code entry
Compiler:  gcc
Compiler Options:
Object Name:  stack.1.1.0.o
Object Host:  pollux

Include Files:

- - - - - - - - - - - - - - - - - - - - -

Filename:  hpc_header.1.1.0.h
Hostname:  pollux
Description:  Initial definition entry
```

```
Owner:  Bonnie M. Edwards
Last Modified:

- - - - - - - - - - - - - - - - - - - - -


------------------------------------------------------------

Module Name:  hpc
Release Number:  1
Revision Number:  1
Variant Number:  0
Is Definition Component:  N
Short Description:  Hyperpascal main module
Long Description:  Main module for the hyperpascal application
Last Modified:  15-APR-95
Owner:  M. Leanne Ward
Status:  UNIT TESTED
Checked Out:  N
Dynamically Configured:  Y
Source Name:  hpc.1.1.0.c
Source Host:  pollux
Source File Description:  Initial code entry
Compiler:  gcc
Compiler Options:
Object Name:  hpc.1.1.0.o
Object Host:  pollux

Include Files:

- - - - - - - - - - - - - - - - - - - - -

Filename:  hpc_header.1.1.0.h
Hostname:  pollux
Description:  Initial definition entry
Owner:  Bonnie M. Edwards
Last Modified:

- - - - - - - - - - - - - - - - - - - - -


============================================================
   ASSOCIATED  OBJECT  LIBRARIES
============================================================



============================================================
   ASSOCIATED  DOCUMENTATION
============================================================


Document Name:  hp_bnf
Release Number:  1
Revision Number:  1
Variant Number:  0
Short Description:  BNF for hyperpascal language
Long Description:  BNF for hyperpascal language
Creation Date:
Last Modified:  15-APR-95
Owner:  Bonnie M. Edwards
Status:  UNDER REVIEW
Checked Out:  N
Dynamically Configured:  N
Source File Name:  /home/grad/edwards/DEV/DOC/hp_bnf.1.1.0.doc
Source File Machine:  pollux
```

```
------------------------------------------------------------


Document Name:  hp_usage
Release Number:  1
Revision Number:  1
Variant Number:  0
Short Description:  Hyperpascal usage documentation
Long Description:  Description and usage documentation for the h
                   yperpascal application.
Creation Date:
Last Modified:  15-APR-95
Owner:  M. Leanne Ward
Status:  UNDER REVIEW
Checked Out:  N
Dynamically Configured:  Y
Source File Name:  /home/grad/edwards/DEV/DOC/hp_usage.1.1.0.doc
Source File Machine:  pollux


============================================================
   APPLICATION  TEAM  MEMBERS
============================================================

Development:
--------------------

M. Leanne Ward            15-APR-95
Bonnie M. Edwards         15-APR-95

Maintenance:
------------------------

Sunder Krishnan           15-APR-95
Bonnie M. Edwards         15-APR-95
```