Software Release Management

André van der Hoek, Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

{andre,rickhall,dennis,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-806-96 August 1996

ABSTRACT

A poorly understood and underdeveloped part of the software process is software release management, which is the process through which software is made available to and obtained by its users. Complicating software release management is the increasing tendency for software to be constructed as a "system of systems", assembled from pre-existing, independently produced, and independently released systems. Both developers and users of such software are affected by these complications. Developers need to accurately document complex and changing dependencies among the systems constituting the software. Users will be heavily involved in the location, retrieval, and assembly process of the systems in order to appropriately configure the software to their particular environment. In this paper we identify the issues encountered in software release management, and present an initial set of requirements for a software release management tool. We then describe a prototype of such a tool that supports both developers and users in the software release management process.

This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 INTRODUCTION

The advent of the Internet and the use of component-based technology have each individually influenced the software development process. The Internet has facilitated geographically distributed software development by allowing improved communication through such tools as distributed CM systems, shared whiteboard systems, and real-time audio and video. Component-based technology has facilitated the construction of software through assembly of relatively large-grained components by defining standards for component interaction such as CORBA [6]. But, it is their combined use that has led to a radically new software development process: increasingly, software is being developed as a "system of systems" by a federated group of organizations.

Sometimes such a process is initiated formally, as when organizations create a virtual enterprise [3] to develop software. The virtual enterprise establishes the rules by which dependencies among the components are to be maintained by the members of the enterprise. Other times it is the connectivity of the Internet that provides the opportunity to create incidental systems of systems. For example, applications are typically built using public-domain software as major components, such as Tcl/Tk [15] for the graphical user interface. The use of public-domain software creates a dependency that, while less formal than in a virtual enterprise, is no less serious a concern.

In either case, component dependencies create a complex system of systems that is in effect being developed by a distributed and decentralized group of organizations. Given that the components of the system are in general pre-existing, independently produced, and independently released systems themselves, the primary issue involves managing the deployment of the system as a whole. In particular, the following two problems become evident.

1. *To developers of a system of systems, deployment of the system is cumbersome.* The developers must carefully and accurately describe their system, especially in terms of its dependencies on other systems. Of course, this problem occurs not just at the outermost level of system construction, but also at all intermediate levels of a hierarchically structured system.

2. *To users of a system of systems, the task of locating, retrieving, and tracking the various components is complicated and error prone.* A consistent set of components must be retrieved from potentially multiple sources, possibly via multiple methods, and placed within the context of the local environment.

These problems lead to a need for what we term *software release management*, which is the process through which software is made available to and obtained by its users. We have defined such a process and built a specialized tool to support that process. The tool, called SRM (Software Release Manager), aids both developers and users in the release process. It is based on two key notions. First, components should be allowed to reside at physically separate sites, yet the location of each component should be transparent to those retrieving the component. Second, the dependencies among components should be explicitly recorded so that they can be understood by users and exploited by the tool. The tool in particular should use dependency information to automate and optimize the retrieval of the components.

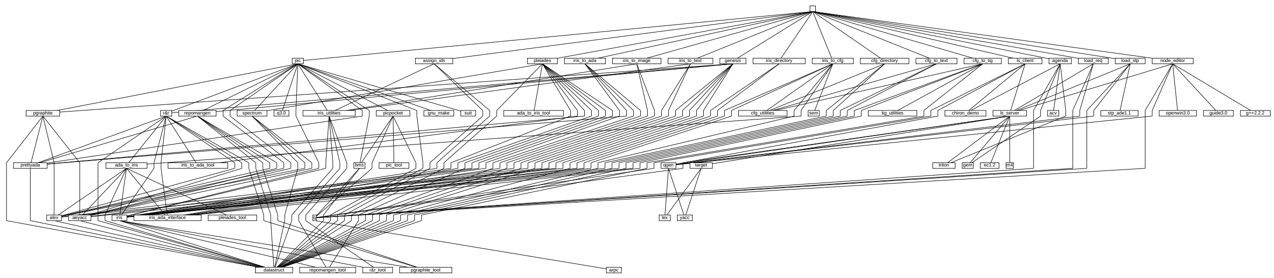

Figure 1: Dependence Graph of the Arcadia Tools.

In this paper we further motivate the need for software release management by illustrating how, in a distributed development setting, a lack of appropriate support for software release management leads to difficulties. We then present a set of requirements for a software release management process and a tool to support that process. Based on these requirements, we describe the functionality of our software release management tool, SRM. We then briefly discuss our experience to date in building and using SRM. We conclude with a look at related work and some directions for the future.

2 A MOTIVATING EXAMPLE

During the past decade, the Arcadia consortium has developed approximately 50 tools to support various aspects of the software development process [10]. Arcadia's research staff is located at four universities across the US. Typically, each tool is developed, maintained, and released by a single university. Many of the tools, however, are dependent on tools developed at other sites. The maintenance of these dependencies has been largely ad hoc, with no common process or infrastructure in place. Figure 1 shows a graph of the dependencies that existed at one point in time. It illustrates the fact that the dependencies among the tools are very complex, and hardly manageable by a human. In fact, the figure presents only a snapshot of the full dependence graph, since it does not show the evolution of the tools into various versions, each with its own dependencies.

Two recent experiences at the University of Colorado clearly show the need for a process and tool to support the management of Arcadia software releases. In the first case, one of the Arcadia tools from a remote site, ProDAG [16], was needed for a project. ProDAG depends on a number of other tools that were also not present at the University of Colorado. Thus, besides ProDAG, each of these other tools and, in turn, the tools upon which they transitively depended, needed to be retrieved and installed. This turned out to be a lengthy and difficult exercise, due to the following reasons.

- *The tools that were needed were distributed from different sites, via different mechanisms.* Typically, FTP sites and/or Web pages were used to provide access to the tools. However, to obtain a different version than the one "advertised", someone responsible for the tool had to be contacted directly.

- *The dependency information was scattered over various sources.* Dependency information for a tool could be found in source files, "readme" documentation files, Web pages, and various other places. Sometimes, dependency information for a single tool could be found in multiple places.

- *The dependency information was incomplete or not even present.* Sometimes the dependency information pointed to a wrong version of a tool, or simply omitted which version of a tool was needed. In other cases, some of the dependencies were not recorded and could only be obtained by explicitly asking the person responsible for the tool. Finding the right person was not always easy because of changes in personnel and responsibilities.

Consequently, retrieving the correct versions of all tools that ProDAG depended on turned out to take far more time, and far more iterations, than expected.

In the second case, another project was initiated at the University of Colorado that also needed ProDAG. However, the version of ProDAG that was installed previously was not usable due to the following reasons.

- *For a number of tools on which ProDAG depends, new versions had been released that fixed a number of important bugs.* Thus, these versions needed to be obtained and installed.

- *The person who initially installed ProDAG at the University of Colorado had left.* Consequently, we did not know which versions of the underlying tools were installed.

Thus, even though most of the dependencies had not changed and the correct versions of most of the tools were already installed, the combination of the above two problems resulted in a complete re-installation of ProDAG. Of course, during the process of obtaining and installing ProDAG anew, the same problems surfaced as experienced during the initial installation.

Analyzing the cause of all the above problems, we can identify at least two critical issues that surface:

- the existence of extremely complex dependencies among components; and

- the lack of user support for location and retrieval of components.

Both are inherent to the software release management problem in cases where software is developed as a system of systems by a geographically distributed group of organizations. Because the example above is becoming ever more common, it should be clear that simply making interdependent components available individually, and retrieving those components individually, does not encourage widespread use of large systems of systems. What is needed is a software release management process that documents the released components, records and exploits the dependencies among the components, and supports the location and retrieval of groups of compatible components.

3 SOFTWARE RELEASE MANAGEMENT

In the past, when a single organization developed a software system, configuration management systems (e.g., Aide de Camp [17], ClearCase [1], and Continuus [2]) were used to support software release management. Once a software system needed to be released, all components of the release were frozen, labeled, and archived in the configuration management system. Through advertising, potential users were made aware of the release, who then had to contact the development organization to obtain the release.

The advent of the Internet has changed this process dramatically. FTP sites and Web pages allow organizations to make their software available to the whole Internet community, to provide information about their products, and even to distribute both free trial versions and licensed revenue versions of their software. In support of the user, there are now search engines, databases, and indexes that provide a way to locate and retrieve a software system over the Internet.

Notwithstanding the success of this approach to releasing software over the Internet, it is insufficient to support the release of systems of systems. New requirements, from both developers and users of such systems, are laid upon software release management. For developers, a software release management process and support tool should provide a simple way to make software available to potential users. This entails the following requirements.

- *Dependencies should be explicit and easily recorded.* It should be possible for a developer to document dependencies as part of the release process, even if dependencies cross organizational boundaries. Moreover, this description should be directly usable by the release management tool for other purposes.

- *A system should be available through multiple channels.* Once a developer releases a system, the release management tool should automatically make it available via such mechanisms as FTP sites and Web pages.

- *The release process should involve minimal effort on the part of the developer.* For example, when a new version of a system is to be released, the developer should only have to specify what has changed, rather than treating the new version as a completely separate entity.

- *The scope of a release should be controllable.* A developer should be able to specify to whom the release is visible and to whom it is not.

- *A history of retrievals should be kept.* This allows developers to track their systems, and to contact users with announcements of new releases, patches, related products, and the like.

For users, a software release management tool should provide an easy way to locate and retrieve components. This leads to the following requirements.

- *Sufficient descriptive information should be available.* Based on this information, a user should be able to easily determine which (versions of) systems are of use.

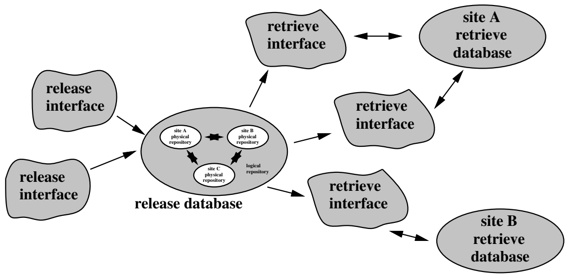

Figure 2: SRM Architecture.

- *Physical distribution should be hidden.* If desired, a user should be unaware of where components are physically stored.
- *Interdependent systems should be retrievable as a group.* A user should be able to retrieve a system and the systems upon which it depends in one step, rather than having to retrieve each system individually, thus avoiding possible inconsistencies.
- *Unnecessary retrievals should be avoided.* Once a system has been retrieved by a user, the release management tool should keep track of this fact, and not re-retrieve the same system if subsequently requested.

A software release management process and support tool that satisfy these requirements will alleviate the problems evident in our motivating example. They will make it easier for developers to release systems of systems, and for users to efficiently obtain those systems in an appropriate configuration.

4 A SOFTWARE RELEASE MANAGER

SRM (Software Release Manager) is a prototype software release management tool that we have developed over the past year. It was designed to explore the issues involved in satisfying the requirements presented in the previous section. SRM is based on two key notions: transparent distribution and the explicit documentation of dependencies. Using these notions, SRM supports both developers and users in the software release management process.

SRM realizes this support through a four-part architecture: a logically centralized, but physically distributed, release database; an interface to place components into the release database; an interface to retrieve components from the release database; and a retrieve database at each user site to record information about the components already retrieved. This architecture is depicted in Figure 2. Below, we discuss each part in detail.

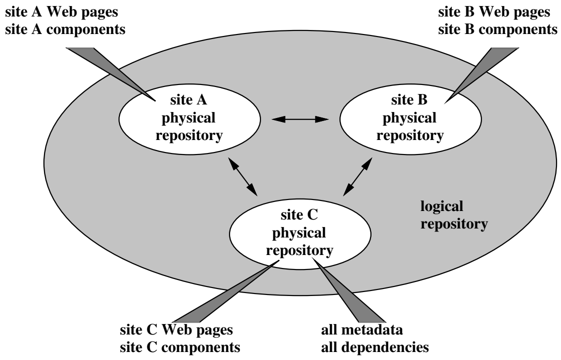

Figure 3: Release Database Structure.

4.1 The Release Database

The release database of SRM has been implemented using NUCM, a distributed repository for versioned artifacts [19]. SRM manipulates NUCM in such a way that the release database is logically centralized, but physically distributed. It is logically centralized in that it appears to users of SRM as if they are manipulating a single database; all artifacts from all distributed sites are visible at the same time. It is physically distributed in that the artifacts are stored in separate repositories spread across different sites. Each site corresponds to a separate organization contributing components to the release database. In particular, when an organization releases a component, a copy of the component is stored in a repository that is physically present at that organization.

Figure 3 illustrates the structure of the release database using a hypothetical arrangement of release information. As we can see, SRM stores four types of artifacts in a release database: the released components, metadata describing each component, the dependency information for the components, and Web pages for each component. A detailed description of these types will be given in later sections. Here we concentrate on the distribution aspects of the release database.

Released components and their corresponding Web pages, which represent the bulk of the data in the repository, are stored at the site where the components were released. In this way, each site provides the storage space for its own components. The other artifacts—i.e., the metadata and the dependency information—are contained in single file that is stored at just one of the sites. This happens to be site C in Figure 3. We have chosen to centralize the storage of metadata and dependency artifacts for simplicity reasons. In the future we plan to explore other, distributed schemes to manage these artifacts.

4.2 The Release Interface

Through the release interface of SRM, developers can *release* a component to the release database, *modify* a release in the release database, or *withdraw* a release from the release database. Releases are provided bottom-up with respect to the dependencies, which is to say that before a component can be released, all other components upon which it depends must have been released.[1] The inverse is true for withdrawing a release. The interface has been implemented using Tcl/Tk [15].

4.2.1 Releasing a Component

To release a component, a developer must provide three pieces of information: metadata describing the component; dependencies of the component on other components; and the source location of the component.

The metadata consists of, among other things, a component name, a component version, contact person for the component, the platform(s) the component runs on, and a detailed description of the component. Based on the metadata information, users that browse the release database can quickly assess the suitability of a particular component.

The second piece of information that a developer must provide describes the first-level, or direct, dependencies of the component on other components. SRM is able to calculate transitive dependencies across multiple components by following paths over first-level dependencies. Figure 4 shows an example of dependency specification. The interface allows for a simple point-and-click selection of first-level dependencies. In this case there are three that have been selected, LPT 3.1, Q 3.3, and TAOS 2.0, highlighted by the dark shading. SRM automatically includes a transitive dependency that it has calculated from previously provided information. In particular, Q 3.3 depends on Arpc 403.4, so this latter system has been highlighted by the lighter shading as a transitive dependency. The combination of the first-level and transitive dependencies is the set of dependencies maintained by SRM for the component being released.

It should be noted that although some of the components might have been released at other sites, specifying them as a dependency for the component being released is as easy as specifying a locally developed component as a dependency; the release database is transparently distributed. For example, TAOS resides at the University of California, LPT resides at the University of Massachusetts, and Q/CORBA resides at the University of Colorado.

The third and final piece of information is the source location of a component. SRM assumes that a component release is contained in a single file, such as a TAR file or ZIP archive. Since SRM makes no assumptions about the format of the file, different formats can be used for different components.

SRM stores all metadata, all dependency information, and a copy of the component itself in the release database. Old versions of the component are not removed; the new release will co-exist with the old release as long as the identifier information is unique. Therefore, the release database

[1] In fact, one could provide releases in any order, although it would not be possible to specify some of the dependencies. The missing dependencies could, and should, be added later using the modify operation.

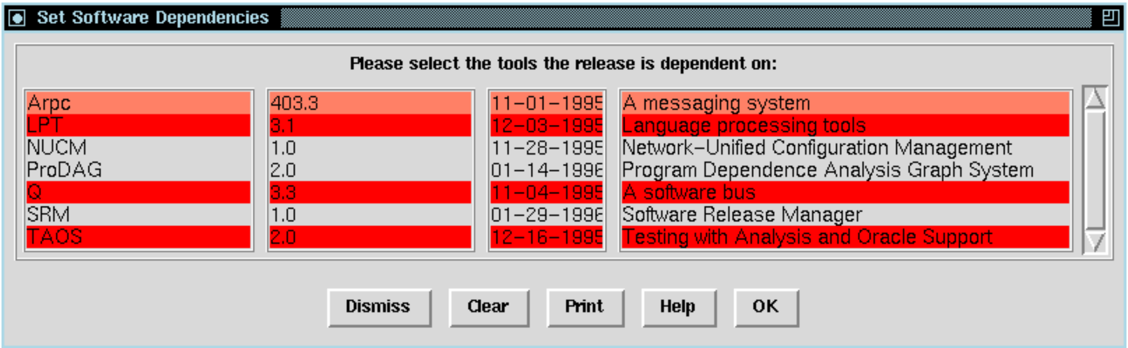

Figure 4: Specifying Dependencies in SRM.

becomes a source of historical information about the various released components. In essence, the SRM repository automatically becomes the documentation for the release management process.

4.2.2 Modifying a Release

SRM allows a developer to modify the information describing a release. One simple reason is that metadata, such as a contact person, may change. A more important reason is to allow underlying dependencies to change. For example, Q/CORBA [13], as mentioned above, depends on Arpc [9]. It happened that a new version of Arpc was created to fix a bug. This fix did not, however, affect the Arpc interface, so no changes to Q/CORBA were required. Once it was determined that Q/CORBA worked properly with the new version of Arpc, and only then, did the Q/CORBA dependency on Arpc get switched to the new version using the modify operation. Notice that a mechanism based on a default dependency, such as "the latest", would not have worked in this scenario. This is because the dependency would have switched automatically from the old to the new *before* it was verified that the new version was compatible.

4.2.3 Withdrawing a Release

The third operation supported by the release interface allows developers to withdraw components from the release database. This functionality is desired for the obvious reasons. Just one restriction is placed on withdrawal of components, but it is an important one that maintains the integrity of the database: the only components that can be withdrawn are those components that no other components are dependent on. For example, since Q/CORBA 3.3 depends on Arpc 403.4, as shown in Figure 4, Arpc 403.4 cannot be withdrawn from the release database. Thus, if a developer indeed wants to withdraw Arpc 403.4, either Q/CORBA 3.3 needs to be withdrawn first, or the dependencies of Q/CORBA 3.3 need to be changed to not include Arpc 403.4.

4.3 The Retrieve Database

At each site that retrieves components using SRM, a retrieve database is maintained. In contrast to the release database, which is shared among sites, the retrieve database is local to each site and only site-specific information is stored there. Every time a component is retrieved by a user at a site, SRM updates the retrieve database at that site. Two pieces of information are important to the retrieve database: metadata and dependencies. Both are obtained from the release database and stored unaltered in the retrieve database upon successful retrieval of a component.

There are two major advantages in administering the metadata and dependencies in a local retrieve database. First, a retrieve database provides users at a site with a central information store about all components installed at that site. In the current situation, without such a central information store, users have to look all over a system environment to figure out which components are present, what functionality is provided by the various components, and what the prerequisites are for the various components to work properly. A central retrieve database alleviates this problem by providing a single place where all needed information is kept. Even if one user retrieves a component, other users can still use the metadata and dependencies from the retrieve database to correctly configure and use the component. The second advantage of the retrieve database concerns the retrieve interface. Since there is a central place to look, the retrieve interface is capable of assessing which components are already present at a site, and can thus avoid retrieval of components that are already present at the site.

Given the central role of the retrieve database, and given our expectation that the retrieve database is going to play a much more important role than simply a data store, one has to be careful in choosing the schema used by the database. In SRM, we have chosen to base our schema on the Management Information Format (MIF) [5] developed by the Desktop Management Task Force (DMTF) as part of the Desktop Management Interface [4]. DMI is a standard framework that is increasingly followed by hardware vendors to administer their installed components. An implementation of the standard allows for plug-and-play interoperability among hardware components. However, DMI is not only targeted towards hardware, but it is also attempting to unify hardware and software administration within a single registry. Since DMI is an emerging standard, we are trying to use MIF as the retrieve database schema. The advantage is that MIF-compatible tools or browsers will be capable of examining which software components are present at a site. We have encountered some shortcomings in MIF, and have begun to enhance it, but discussion of that topic is beyond the scope of this paper.

4.4 The Retrieve Interface

Once components have been placed into the release database, the retrieve interface of SRM can be used to retrieve the components from the database. To do so, SRM uses information from both the release and retrieve databases to support a user in locating and retrieving components. In effect, the retrieve interface forms a bridge between the development environment and the user environment.

We have experimented with two retrieve interfaces: a dynamic, Java-based interface, and a static, Web-page-based interface. We discuss each below.

4.4.1 The Java Retrieve Interface

Developing the retrieve interface in Java [18] has two significant advantages. First, the connectivity of the Internet guarantees widespread access to the release database. Standard Internet browsers, such as Netscape Navigator and Microsoft Explorer, can be used to retrieve components.[2] Second, Java applets are inherently *active*, which means that they are able to execute at the client site. This dynamism allows real-time interaction with both the user and the retrieve database.

When users want to retrieve a component, they use the Java interface, which can be obtained through an Internet browser. The browser executes the Java interface applet, allowing a user to examine the components that are available in the release database. While browsing, both the metadata describing the various components and the dependencies among the components are displayed. This allows a user to quickly assess the suitability of the various components.

Once a component is selected, its dependencies are presented to the user. The user can then manually select which of the dependent components should be retrieved. By examining the local retrieve database, SRM will have pre-selected the dependencies not already present at the site. Users can override these defaults, and choose whatever subset of dependencies is desired. During this process of selecting components, SRM automatically turns off or on, as appropriate, the transitive dependencies emanating from the selected components. This further simplifies the selection process for the user and minimizes the set of components to retrieve. Typically, however, there is no need for a user to manipulate the dependencies, because the dependencies marked by SRM constitute the minimum set needed to obtain a complete system at the local site. Even as other users retrieve components, this will be the case, since the central retrieve database administers all retrieved components at that site.

When a user decides to retrieve a component together with some subset of its dependencies, the Java interface uses a CGI script [14] to contact the underlying NUCM release database. The CGI script retrieves the selected components from the various distributed sites and ships them back to the Java applet. The applet then stores the components at the local site, updating the retrieve database with the metadata and dependencies of the newly obtained components.

It should be noted that a user of the Java interface is not aware of the fact that the various components might have originated from several geographically distributed sites. The distribution is hidden by both the Java interface, which lists all components irrespective of where they originate, and the CGI script, which silently retrieves the components from the various repositories and ships them back to the user.

[2] Ideally, this would happen transparently, but due to security restrictions, the downloaded Java interface applets currently have to be executed by a spawned browser.

4.4.2 The Web Page Interface

The Web page interface was designed to provide an alternative to the Java interface. We need this alternative for two reasons. First, and most importantly, the Java interface requires a retrieve database, and it is expected that not every user will want or need such a solution. Second, Java is still seen as a major security risk by many organizations and, consequently, execution of "foreign" Java applets is often restricted. Thus, besides creating the dynamic Java interface, we built a static, Web-page-based interface. Every time a component is released, SRM creates a Web page for that component and updates a main Web page listing all available components.

When users want to retrieve a component, they first retrieve the main Web page through their Internet browser. Upon selection of a component from the main Web page, the Web page corresponding to the selected component will be presented to the user. The contents of this page is the metadata that were provided upon release of the component. In addition, the Web page shows the dependence graph for the component and provides selection buttons to turn off or on dependent components for retrieval. This portion of the interface is shown in Figure 5. In this example, ProDAG is being retrieved, and three of its four dependencies have been selected for retrieval as well.

Due to the fact that Web pages are static in nature, SRM is not able to use the dependency information to turn off or on transitive dependencies on behalf of the user, nor is it able to manipulate the retrieve databases at local sites. Instead, the user will have to perform dependency selection manually by examining the dependence graph shown in the page and pressing the appropriate buttons. Other than this the Web page interface is equivalent to the Java interface. It allows for the retrieval of complete systems at once, using standard Internet browsers, while hiding distribution from the user.

5 REQUIREMENTS EVALUATION

SRM as it currently stands covers, to a greater or lesser extent, all of the requirements for software release management that were enumerated above. In particular, it hides distribution and promotes the use of dependency information. However, it satisfies some of the other requirements only partially. Specifically, for SRM to fully satisfy all requirements, it needs the following extra functionality.

- *Specification of external dependencies.* It cannot be expected that all components of a system are released to SRM. Typically, some of the components will be under the control of organizations not employing our tool. One should nevertheless be able to manage such external dependencies using SRM. We believe that if they are specified as a list of external Web pages or FTP sites, it will be straightforward for SRM to deliver a complete system to a user.

- *Annotation of dependencies.* Not all dependencies are alike. For instance some dependencies are run-time dependencies, while others are build-time dependencies. If developers were able to easily describe or annotate dependencies during release, these dependency annotations could be used to better understand what is required to build, install, and/or execute a system.

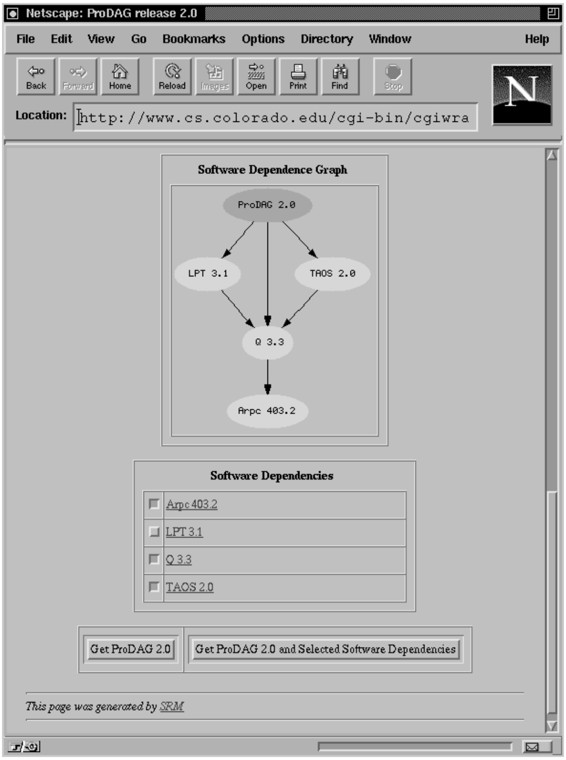

Figure 5: Portion of an SRM Retrieval Web Page.

- *Access control.* Currently no access control is provided in SRM. A typical software development scenario involves different levels of release and different groups of users. For example, many organizations like to distinguish between alpha, beta, and full releases. Ideally, they would like to control to whom those different kinds of releases are made available. Clearly, some sort of access control mechanism is needed.

- *Publication of software releases through additional channels.* Currently, SRM only publishes available software over the Internet via either a Java or a Web page interface. These mechanisms need to be complemented with several other means of publishing a release, such as FTP sites, a developer-controlled mailing list, automatic postings to one or more news groups, and the like.

Despite its current shortcomings, SRM clearly has advantages over the “label and archive” paradigm traditionally used for software release management. We have discussed how SRM improves both the process by which developers make components available to users and the process by which users obtain the components. Besides these two important improvements, however, SRM has another advantage.

In settings where a virtual enterprise is created by a group of organizations, the old and often ad hoc release management process employed by the various participating organizations can be replaced by a single, more disciplined, and unified process employed by all the organizations. Each of the organizations is still able to use its own development process, its own configuration management system, and its own development tools. But the various mechanisms for release management can be unified under SRM to provide a common point of intersection for the organizations. In this way the various organizations are flexibly joined at a high level of abstraction. SRM provides developers a basis for communicating about interdependent components that avoids low-level details of path names, FTP sites, and/or URLs. Moreover, because SRM maintains all versions of all components in its release database, and maintains the status of components located at each site in its retrieve databases, it becomes the language and mechanism for inter-organization communication about interdependent components.

6 EXPERIENCE

SRM has been implemented using a wide variety of technologies, including Java, MIF, HTML [12], and Tcl/Tk. In addition, the underlying repository, NUCM, has been implemented as a CORBA object using the Q/CORBA system [13]. Naturally, putting these technologies together and deploying the resulting system turned out to be a non-trivial exercise. In fact, because some of the technologies are new, using them for a complex task such as SRM revealed significant problems for them in their own right. Below, we discuss the major problems we encountered in implementing SRM and then present our initial experiences in deploying SRM.

6.1 Implementation Issues

The first problem we encountered in building SRM regarded the interfacing of Tcl/Tk and CORBA. All of the release interface has been implemented using Tcl/Tk. Since no Tcl/Tk-to-CORBA communication package is yet available, SRM communicates with the release database using a series of small interface programs. Each program is a direct implementation of one of the NUCM functions specified in its CORBA interface. Each program's arguments are textual representations of the input parameters of the interface function it represents, and its output is a textual representation of the output of the interface function. Thus, for each interface function that Tcl/Tk needs to access in the NUCM repository, it has to execute the corresponding interface program. Clearly, this is a very expensive process that slows down the response time of SRM considerably.

The second implementation problem we encountered regarded the Web page and Java interfaces. Both the Web pages and the Java programs are generated when a component is released, because generating them once at release time is much more efficient than generating them every time one of them is needed. Both the Web pages and the Java programs are stored in the release database, so that each can be used from all sites. To access them from a standard Internet browser, we had to use CGI scripts [14] to retrieve them from the database and return them to the browser. The structure of, and logic used in, these CGI scripts is simple as long as a single artifact has to be returned to a browser. But in the case of the Web interface, the Web page for a component refers to the corresponding dependence graph, and thus two artifacts have to be returned. Similarly, in the case of the Java interface, multiple classes have to be returned. This multiple return problem complicates CGI scripts considerably, because some sort of caching needs to be implemented to facilitate return of multiple artifacts at the same time.

CGI scripts are the source of another problem. At the University of Colorado, the scripts execute in a protected environment due to security concerns. Since such an environment is not as of yet standardized, it is impossible to generalize a scripting solution. Therefore, as part of installing SRM itself, the CGI scripts will sometimes need to be tailored to adapt to a site's specific approach to security.

We encountered another problem related to security. Firewalls turned out to be a major challenge to overcome in deploying our underlying NUCM release database technology over a wide area network. The commercial CORBA implementation that we used to create NUCM was not able to pass requests through the firewall that is in place at the University of Colorado. Consequently, we had to reimplement NUCM using the Q/CORBA system, which supports communication through firewalls by using a special portmap program.

The final problem we encountered in implementing SRM was the fact that the MIF standard is not a good one in which to adequately describe software components. Because of its hardware orientation, many of the required attributes are superfluous, while others that would be useful and appropriate are simply missing. Again, a full discussion of the issues is beyond the scope of this paper.

6.2 SRM Deployment Experience

Currently, SRM is in use as the release management system for the software produced by the Software Engineering Research Laboratory at the University of Colorado.[3] Our experiences with initial versions of SRM in this setting have allowed us to evaluate the user interface and functionality provided. Based on the feedback received, many modifications and enhancements have been made to both the interface and functionality, resulting in the system as presented in the previous sections.

In addition, SRM is currently being deployed at all universities participating in the Arcadia project. In particular, colleagues at the University of California, Irvine, have set up an SRM repository for their own use. Initial tests in connecting this repository with the one in Colorado, and thus using the distributed features of SRM to the fullest, have been extremely successful. Therefore, we shortly expect SRM to be in use across the Arcadia project, and to make SRM available to the general public thereafter.

[3] See http://www.cs.colorado.edu/users/serl for a pointer to our released software.

7 RELATED WORK

To date, software release management has received very little attention from either the academic or the commercial world. There have been only two systems that have directly addressed software release management: `ship` [11] and the FreeBSD porting system [8].

AT&T's `ship` is an extensive release management system that is strongly tied into a proprietary software reuse architecture. Unlike SRM, which provides control to both developers and users, `ship` places all control in the hands of the developer. A developer decides to which sites a release is to be shipped, and then `ship` takes control of the various user sites and installs the necessary software components there. While doing so, `ship` maintains small databases of components and versions installed at the user sites. Using the information in these databases, `ship` can check for necessary components being present at the user sites, and report back to the developer site about whether the installation can be completed or not.

Despite its power, `ship` still suffers from similarities to the old label and archive paradigm: the developer is in control, not the user. Moreover, the format used for the local databases is proprietary, and therefore there is no relation to software retrieved from other sites. Further complicating the use of `ship`are the severe restrictions placed on the development of the software to be released. In particular, the software needs to be developed within a strictly defined hierarchy of directories and must make use of the proprietary tools associated with the reuse architecture.

The FreeBSD porting system supports the FreeBSD user community by organizing freely available software into a carefully constructed hierarchy known as the "ports collection". The system uses specialized `make` [7] macros and variables to enable the building of systems in the hierarchy as well as to manage their associated dependencies. It uses various forms of heuristics to determine a site's state and employs the results in building and installing a software package. SRM seems to be complementary to the FreeBSD porting system. Whereas the emphasis of SRM is on making software systems available to users and facilitating users in obtaining such software systems, the emphasis of FreeBSD is on the build and installation process. It is conceivable that the two systems could cooperate closely to support a user beyond the point where SRM delivers a set of components. FreeBSD would be responsible for receiving the components delivered by SRM and then building and installing the components for the user.

Some other systems partially support software release management, but again, they seem to be complementary to SRM. For example, configuration management systems typically have a facility to create a software release out of the sources present in the configuration management system, but do not make such a release available to users. In other cases, software is distributed with an installation agent that examines the local environment for the necessary components. However, such agents do not pay attention to the way the software is distributed to a user site. So, although there is support on either side of the functionality provided by SRM, there seems to be no system like SRM that specifically sets out to bridge the gap between.

8 CONCLUSIONS

The work described here represents a novel approach to the software release process. By means of a software release management system, SRM, a bridge has been built between the development process and the deployment process through which both developers and users are supported. To developers, SRM provides a convenient and uniform way to release interdependent systems to the outside world. To users, SRM provides a way to retrieve these systems through the well-known interface of the Internet. For both, the fact that components are released from various geographically distributed sources is hidden.

Even though the basic idea behind SRM is simple—to provide distribution transparency to the release of interdependent software components—its fundamental contribution—the awareness and support for both the development *and* the deployment processes—is an important one. We believe much more research is needed to support both at the same level, and intend to use SRM as a starting point in this research.

Our future plans call for several significant functionality improvements to SRM, including the accommodation of external dependencies, a mechanism for annotating dependencies, and access control over releases. In addition, we plan to investigate how different SRM repositories can be federated to form more flexible hierarchies of organization.

9 ACKNOWLEDGMENTS

This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

REFERENCES

[1] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. ClearCase MultiSite: Supporting Geographically-Distributed Software Development. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, 1995.

[2] Continuus Software Corporation, Irvine, California. *Continuus Task Reference*, 1994.

[3] W.H. Davidow and M.S. Malone. *The Virtual Corporation.* Harper Business, 1992.

[4] Desktop Management Task Force. *The Desktop Management Interface.* Available on the world wide web at *http://www.dmtf.org:80/techlinks/white_papers.html.*

[5] Desktop Management Task Force. *The Desktop Management Interface and the Management Information Format.* Available on the world wide web at *http://www.dmtf.org:80/techlinks/white_papers.html.*

[6] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification, Version 1.2.* Object Management Group, Framingham, Massachusetts, December 1993.

[7] Stuart I. Feldman. Evolution of Make. In *Proceedings of the International Workshop on Software Versioning and Configuration Control*, pages 413–416, 1988.

[8] FreeBSD Documentation Project. *FreeBSD Handbook.* Available on the world wide web at *http://-www.freebsd.org/handbook/handbook.html.*

[9] Dennis Heimbigner. Arpc: An augmented remote procedure call system. Technical Report CU-ARCADIA-100-96, University of Colorado Arcadia Project, Boulder, CO 80309-0430, Revised 19 June 1996. Version 403.4.

[10] R. Kadia. Issues Encountered in Building a Flexible Software Development Environment. In *SIGSOFT '92: Proceedings of the Fifth Symposium on Software Development Environments*, pages 169–180. ACM SIGSOFT, December 1992.

[11] B. Krishnamurthy, editor. *Practical Reusable UNIX Software*, chapter 3, pages 91–120. John Wiley & Sons, Inc., New York, 1995.

[12] Massachusetts Institute of Technology/World Wide Web Consortium. *Hypertext Markup Language – 2.0*, 22 September 1995. Available on the world wide web at *http://www.w3.org/pub/WWW/MarkUp/-html-spec/html-spec_toc.html.*

[13] M.J. Maybee, D.M. Heimbigner, and L.J. Osterweil. Multilanguage Interoperability in Distributed Systems. In *Proceedings of the 18th International Conference on Software Engineering*, pages 451–463. Association for Computer Machinery, March 1996.

[14] National Center for Supercomputing Applications. *The CGI Specification.* Available on the world wide web at *http://hoohoo.ncsa.uiuc.edu/cgi/interface.html.*

[15] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, 1994.

[16] D.J. Richardson, T.O. O'Malley, C.T. Moore, and S.L. Aha. Developing and Integrating ProDAG in the Arcadia Environment. In *SIGSOFT '92: Proceedings of the Fifth Symposium on Software Development Environments*, pages 109–119. ACM SIGSOFT, December 1992.

[17] Software Maintenance & Development Systems, Inc, Concord, Massachusetts. *Aide de Camp Configuration Management System*, April 1994.

[18] Sun Microsystems Computer Corporation. *The Java Language Specification*, 11 May 1995.

[19] A. van der Hoek, D. Heimbigner, and A.L. Wolf. A Generic, Peer-to-Peer Repository for Distributed Configuration Management. In *Proceedings of the 18th International Conference on Software Engineering*, pages 308–317. Association for Computer Machinery, March 1996.