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

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 preexisting, 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.

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 [5]. 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 [4] 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 [13] 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 in effect is 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), is currently in use at the University of Colorado.

SRM is based on two key notions. First, components are allowed to reside at physically separate sites, yet the location of each component is transparent to those using SRM. Second, the dependencies among components are explicitly recorded so that they can be understood by users and exploited by the tool. The tool in particular uses dependency information to automate and optimize the retrieval of the components.

Software release management fits within the larger context of software configuration and deployment, which involves additional tasks such as installation, update, and removal. This larger context is being investigated in our Software Dock project [8]. SRM serves as the release manager in the Software Dock prototype, but is designed to be used independently.

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. Based on these requirements, we describe the functionality of our software release management tool, 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.

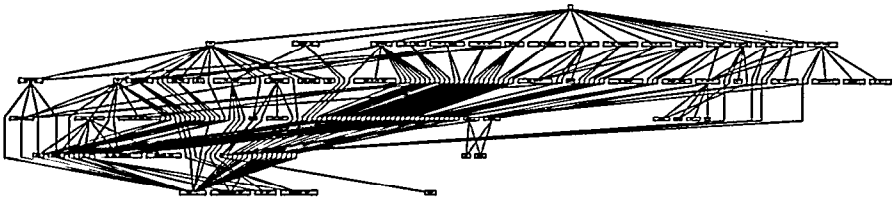

Fig. 1. Dependence Graph of the Arcadia Tools.

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 [14], 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 complex dependencies among components; and
- the lack of user support for location, retrieval, and tracking of components.

These problems are not unique to the situation in which the Arcadia consortium finds itself. Instead, these problems are fundamental and common to many other instances of software development. As examples, contracting often involves many contractors who independently develop many components constituting one or more software products that are put together by an integrator; large companies often consist of many independently functioning departments, each supported by their own set of software development tools; and companies sometimes form virtual enterprises to create a joint product, assembled from components developed by each company. In all these cases, the relative independence of the participating groups, combined with the need to produce a joint result, creates a situation similar to the one presented above. Components have to be released among groups, systems have to be released to external organizations, many versions of many different components are produced, and, most importantly of all, dependencies among components become increasingly complex.

From the above discussion it should be clear that software release management is an important part of the software development process. Simply making available and retrieving interdependent components individually neither facilitates independent software development nor encourages 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 [16], ClearCase [1], and Continuus [3]) 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, and under what conditions, the release is visible. Licensing, access control, and electronic commerce all need to seamlessly interoperate with the software release management process.
- *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.

- *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, and employs a process similar to the traditional "label and archive" paradigm described in the same section. It assumes an archive containing a release is made available to users, and that users are responsible for locating and retrieving these archives. However, SRM enhances this process in two important ways.

- It *structures* the information used in the release management process, and then *uses* this structure in automating much of the process to support developers and users.
- It *hides* physical distribution from its users. In particular, dependencies among components can spread across multiple, physically distributed, organizational boundaries, and both developers and users are capable of accessing the information regarding components in a way transparent to the location of the information.

SRM realizes this process 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 the first three parts of the architecture in detail. The fourth part, the retrieve database, has been superseded by the Software Dock [8] and is not discussed further here.

4.1 The Release Database

The release database is a repository that SRM uses to store the structured information pertaining the releases, as well as the releases themselves. The database

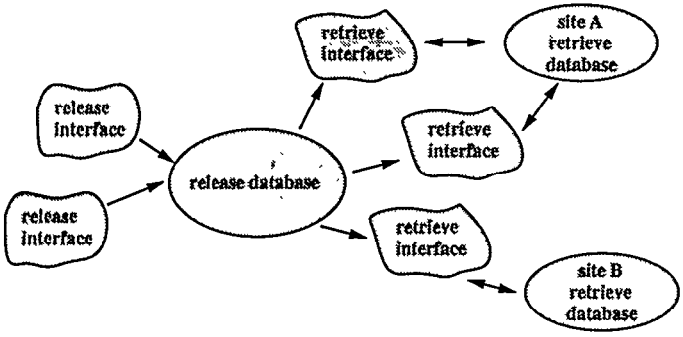

Fig. 2. SRM Architecture.

has been implemented using NUCM, a distributed repository for versioned artifacts [18]. 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. 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

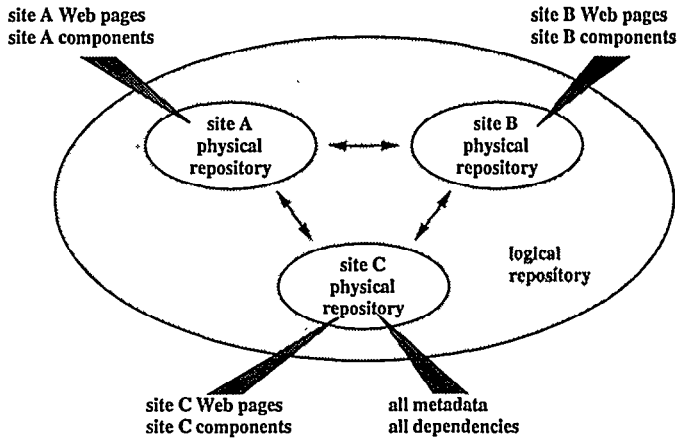

Fig. 3. Release Database Structure.

other components upon which it depends must have been released.[1] The inverse is true for withdrawing a release.

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.

[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.

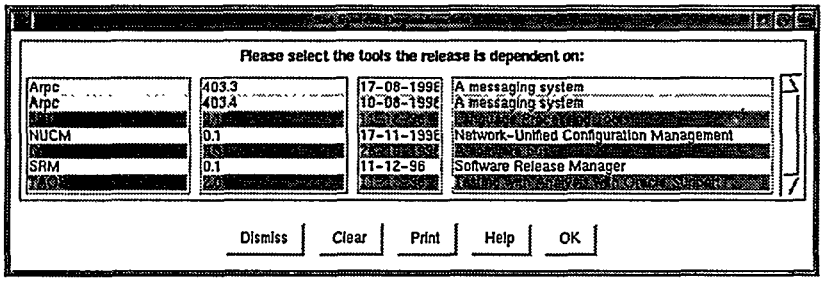

Fig. 4. Specifying Dependencies in SRM.

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 2.0 resides at the University of California, LPT 3.1 resides at the University of Massachusetts, and Q 3.3 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. Therefore, the release database becomes a source of historical information about the various released components and their dependencies. In essence, the SRM repository automatically becomes the documentation for the release management process.

One assumption made throughout this process is that components are released through SRM only. However, there may very well be dependencies on components that have been produced by organizations that do not use SRM. For such cases, SRM provides the concept of *foreign* dependencies. A developer who wants to explicitly state that a component is dependent on a foreign dependency (which is the case for SRM itself, since it is built using Tcl/Tk [13]) simply releases two components. First the developer releases Tcl/Tk to SRM, then the developer releases the component that needed to be released in the first place. In both cases the same process is followed, and in both cases the result is the same: a component stored inside SRM that can be used as a dependency and that can be retrieved by users. However, there is one important difference. Instead of providing an archive containing the release, for foreign dependencies a developer specifies a URL through which the component can be retrieved. As opposed to storing the component inside SRM, SRM will maintain and use the given URL transparently to retrieve the component when needed.

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 3.3 [12], as mentioned above, depends on Arpc 403.4 [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 3.3 were required. Once it was determined that Q 3.3 worked properly with the new version of Arpc, and only then, did the Q 3.3 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.

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 upon which there are no dependencies.[2] For example, since Q 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 3.3 needs to be withdrawn first, or the dependencies of Q 3.3 need to be changed to not include Arpc 403.4.

4.3 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. SRM uses information in the release database 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.

The retrieve interface is built as a static, Web-page-based interface because 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.

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

[2] This rather strong policy can be circumvented by simply modifying the release to have no contents. This crude mechanism maintains the integrity of the database at the same time as it provides developers the flexibility to remove dependent systems, but should only be used in extreme circumstances.

user. The contents of this page is the metadata that were provided upon release of the component. This allows a user to quickly assess the suitability of the various components. 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.

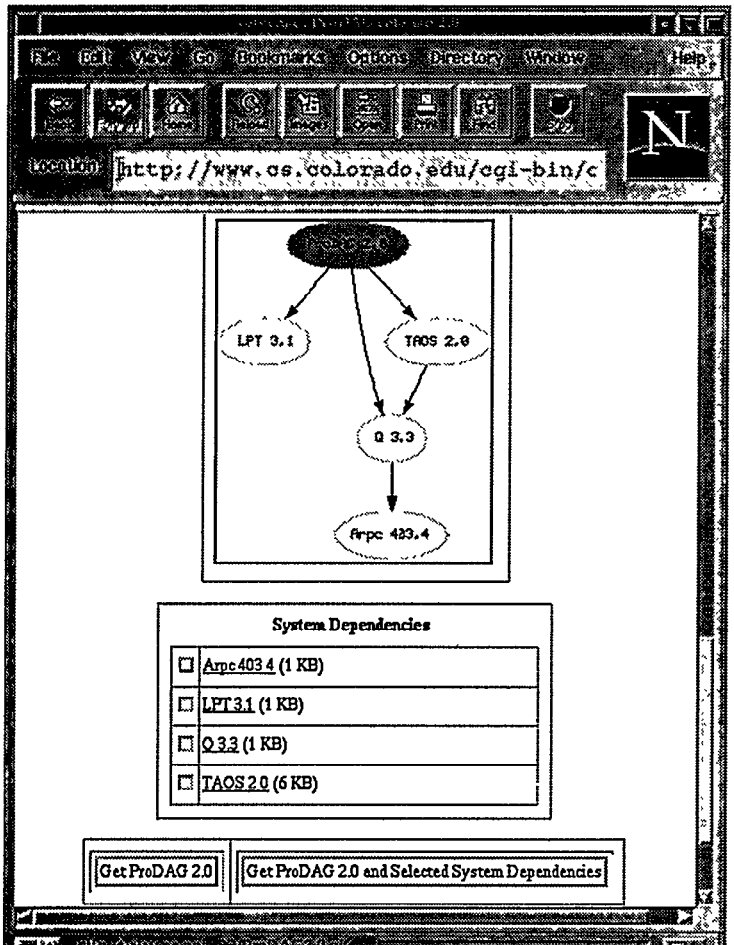

Fig. 5. Portion of an SRM Retrieval Web Page.

A user of the retrieve interface is not aware of the fact that the various components might have originated from several geographically distributed sites. The

distribution is hidden within the SRM release database, which silently retrieves the components from the various repositories and ships them back to he user.

Finally, it should be noted that the retrieve interface keeps detailed statistical information about the number of retrievals that have taken place, allowing developers to assess the usage of their components.

5 Requirements Evaluation

The current version of SRM covers, to a greater or lesser extent, the requirements for software release management enumerated in Section 3. Most importantly, it hides distribution and promotes the use of dependency information.

However, we can see places where SRM can be strengthened in order to fully satisfy the requirements.

- *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 (and even automate) what is required to build, install, and/or execute a system.
- *Ranges of dependencies.* SRM at the moment only supports dependencies on a single version of a component. In the real world, one often would like to specify ranges of compatible versions as opposed to single version dependencies.
- *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.
- *Licensing mechanisms.* Often software is not just released to users, but users are required to electronically sign a license agreement. SRM should provide a set of standard license templates, as well as an escape capability that allows it to hook into any proprietary licensing mechanism.
- *Publication of software releases through additional channels.* Currently, SRM only publishes available software over the Internet via a Web-page-based 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.

Avoiding unnecessary retrievals is the one requirement that SRM does not address on its own. Instead, it satisfies this requirement through an integration with the Software Dock, which we describe elsewhere [8].

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 set of semi-independent groups is cooperating to build one or more software products, such as in all of the cases discussed in Section 2, the old and often ad hoc release management process employed by the various participating groups can be replaced by a single, more disciplined, and unified process employed by all the groups. Each of the groups 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 groups 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 intergroup communication about interdependent components.

6 Related Work

To date, software release management by itself has received very little attention from either the academic or the commercial world. Many of the existing software deployment systems do in fact incorporate some rudimentary form of software release management, but only because it is needed to support their core activities, such as configuration and installation. However, none of these systems have specifically and methodologically tackled the issues of software release management and, therefore, contain deficiencies with respect to software release management. Specifically, most of the deployment systems lack adequate dependency handling, while the ones that do tend to lack distribution. Enhancing these systems with the features explored in SRM, which is the only existing system that fully addresses release management, would result in much more useful and powerful deployment tools.

Below, we discuss a representative sample of the existing deployment systems and relate these systems to SRM.

6.1 20/20's NetInstall

NetInstall [15] is a system that provides automatic installation of software that is published on Web pages. When a user selects a software system, an installation agent is downloaded that installs the desired software system for the user. Although different in focus from SRM, it is relevant because implicit in NetInstall is a release "builder" that creates the appropriate installation agent. Through this release builder the developer is supported in creating a release, while the user is supported by automatic retrieval and installation of the release.

NetInstall differs substantially from SRM in that it focuses on single system delivery, whereas SRM manages the much more complex problem of coordinated releases of systems of systems. In addition, NetInstall does not support publication of the release (i.e., developers have to create Web pages containing installation agents for their systems themselves).

6.2 Marimba's Castanet and Open Software Associates' netDeploy

Marimba's Castanet [11], as well as Open Software Associates' netDeploy [2], provides for automatic software updates once a system has been installed on a site. In essence, these systems mirror software from a single distribution site onto a set of mirror sites, keeping the mirror sites up to date with the distribution site. Both systems provide a "tuner" through which users can select the software that they want to have mirrored onto their site.

In many ways Castanet and netDeploy are similar to SRM; however, there are some important differences. Both systems, similarly to NetInstall, focus on single system settings and neither system is therefore capable of managing dependencies or relating components developed at different sites the way SRM does. In addition, components are continuously and automatically kept up to date by Castanet and netDeploy. Given that new versions of software can be incompatible with existing versions, this can be a frustrating experience for users who depend on a particular version of a software system. SRM will prevent this from happening by placing the user in control of when a new release is retrieved and thus allowing a user to judge, based on the information presented, whether to update the software or not.

6.3 AT&T's ship and Tivoli's TME

AT&T's ship [7] (and similarly Tivoli's TME [17]) is an extensive software deployment system that provides for automatic software installation and update. 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.

While clearly being more powerful than Castanet and netDeploy, ship still does not address all software release management issues. It suffers from similarities to the old label and archive paradigm: the developer is in control, not the user. Finally, even though ship is capable of handling dependencies, it is only capable of handling dependencies that stem from the same release site. As each release site corresponds to a single organization, this places a serious limitation on the usability of ship that SRM does not possess.

6.4 FreeBSD Porting System

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 [6] 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 is 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.

7 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.

The basic concept behind SRM is simple: to provide distribution transparency to the release of interdependent software components. However, its fundamental contribution—the awareness and support of the release management process—is an important one. The software release management process has largely been ignored in the past, but its identification as a major component in the deployment process has led us to develop SRM, a system that better supports the actual process taking place. The key differentiators that provide this support are the following.

- *SRM uses structured information about the release management process to automate much of this process.*
- *SRM hides distribution and decentralization from both developers and users.*
- *SRM places users in control of when deployment of a software system takes place.*

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

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

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.

Our future plans call for several significant functionality improvements to SRM, including the incorporation of standard licensing templates, 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. Finally, we are looking into how SRM can be integrated with other software deployment systems.

Acknowledgments. We thank Jonathan Cook for being a willing first user of SRM, and would like to acknowledge Antonio Carzaniga, Mike Hollis, and Craig Snider for their contributions to the design and implementation of SRM.

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. Open Software Associates. OpenWEB netDeploy. Available on the world wide web at *http://www.osa.com/products/openweb/oweb000.htm.*
3. Continuus Software Corporation, Irvine, California. *Continuus Task Reference*, 1994.
4. W.H. Davidow and M.S. Malone. *The Virtual Corporation.* Harper Business, 1992.
5. 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.
6. S.I. Feldman. Evolution of Make. In *Proceedings of the International Workshop on Software Versioning and Configuration Control*, pages 413–416, 1988.
7. G. Fowler, D. Korn, H. Rao, J. Snyder, and K.-P. Vo. Configuration Management. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 3. Wiley, New York, 1995.
8. R.S. Hall, D.M. Heimbigner, A. van der Hoek, and A.L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *Proceedings of the 1997 International Conference on Distributed Computing Systems*, pages 269–278. IEEE Computer Society, May 1997.
9. D.M. 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. Marimba. Castanet White Paper. Available on the world wide web at *http://-www.marimba.com/developer/castanet-whitepaper.html.*
12. 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.
13. J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, 1994.
14. 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.
15. 20/20 Software. 20/20 NetInstall. Available on the world wide web at *http://-www.twenty.com/Pages/NI/NI.shtm.*
16. Software Maintenance & Development Systems, Inc, Concord, Massachusetts. *Aide de Camp Configuration Management System*, April 1994.
17. Tivoli. Tivoli TME 10 Software Distribution. Available on the world wide web at *http://www.tivoli.com/products/Courier/.*
18. A. van der Hoek, D.M. 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.