SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:77–98 (DOI: 10.1002/spe.496)

Software release management for component-based software

André van der Hoek[1,*,†] and Alexander L. Wolf[2]

[1]*Institute for Software Research, Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92612–3425, U.S.A.*
[2]*Software Engineering Research Laboratory, Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309, U.S.A.*

SUMMARY

Software release management is the process through which software is made available to and obtained by its users. Until now, this process has been relatively straightforward. However, the emergence of component-based software is complicating software release management. Increasingly, software is constructed via the assembly of pre-existing, independently produced, and independently released components. Both developers and users of such software are affected by these complications. Developers need to accurately document the complex and changing dependencies among the components constituting the software. Users must be involved in locating, retrieving, and assembling components in order to appropriately bring the software into their particular environment. In this paper, we introduce the problem of release management for component-based software and discuss SRM, a prototype software release management tool we have developed that supports both developers and users in the software release management process. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: software release management; components; component-based software; distributed development; software deployment

1. INTRODUCTION

In the world of component-based software development, software systems are created by integrating independently developed, pre-existing components[‡]. Many different components may be offered in a common marketplace by many different organizations. These organizations, located all around the globe, continuously improve the components and independently release new versions of them as they make these improvements [1–3].

*Correspondence to: André van der Hoek, Institute for Software Research, Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92612–3425, U.S.A.
†E-mail: andre@ics.uci.edu

‡In this paper, we adopt the common view of a component as a relatively large-grained, mostly self-contained, and independently identified part of one or more software systems that is deployed as a single, coherent unit [1].

Contract/grant sponsor: Defense Advanced Research Projects Agency (DARPA)
Contract/grant sponsor: Air Force Research Laboratory; contract/grant numbers: F30602-00-2-0599; F30602-00-2-0608

Received 24 August 2001
Revised 16 August 2002
Accepted 1 October 2002

Published online 27 November 2002

Much research and development effort is currently being expended on creating technologies to support component-based software development. Among them are component platforms (e.g. .NET [4], EJB [5], Koala [6]), techniques for the predictable assembly of components [2], and enhancements to existing programming languages to incorporate modeling constructs for specifying components (e.g. ArchJava [7], Acoel [8]).

Far less attention has been paid to the question of what happens after development: how should developers release component-based software and how can users subsequently obtain such software—especially if the software under consideration consists of many components that are released by many organizations at many different geographical locations? Typically, the burden rests either with the user of the software, who has to scour the Internet to locate and retrieve the necessary set of components, or with the developer of the software, who has to create an archive containing all necessary components. In either case, a manual process is followed that is often labor intensive, error prone, and not very flexible.

We term this problem *software release management*, which is the process through which software is made available to, and obtained by, its users [9]. At the heart of software release management is the notion of *dependence*, by which we refer to a component's need for a set of other components in order for that component to properly function. In other words, if a user were to only obtain the component itself, or the component and only a subset of its dependent components, the resulting software would be incomplete and not operate. Understanding a component's dependencies is complicated by the fact that components may be developed by different organizations, that those organizations autonomously control the release of new versions of their components, that each version of a component may have different dependencies, and that dependent components may themselves be complex component-based systems. Therefore, documenting dependencies accurately, and then using the documentation to both drive and constrain software release management, is critical for supporting developers and users of component-based software.

We have defined a flexible release management process and built a specialized tool to support that process in the context of distributed, component-based software development. The tool, called Software Release Manager (SRM), is based on two key notions. First, while components can be released from physically separate sites, the actual location of each component is transparent to those using the SRM. Second, dependencies among components are explicitly recorded so that they can be understood and exploited by the tool and its users. In particular, the tool helps developers automatically document and track transitive dependencies, and helps users in retrieving not just components, but also all of the dependent components.

An important assumption we are making is that, while SRM is responsible for storing and reasoning over dependencies, it itself is not responsible for generating or otherwise deriving the dependencies. Instead, when a developer releases a component, they (or some tool used by them) are expected to specify the dependencies of that component. SRM is therefore independent of any specific component platform such as .NET or EJB. This limits the functionality of SRM, in that it cannot exploit specialized features of the component platform (for example, to determine the consistency of a component-based system or to automatically calculate the dependencies). However, it significantly broadens its applicability.

SRM is focused specifically on the activities that take place between the time when components are developed and when they are installed. SRM intentionally does not support traditional configuration management of source code, nor does it support the installation, configuration, activation, or run-time

reconfiguration of a component-based application. Instead, SRM forms a bridge from the organizations where components are authored and released, to the organizations where the components are assembled into an application. Although some existing release tools incorporate aspects of this functionality (as discussed in Sections 2 and 6), we argue why they fall short and how a tool such as SRM fills an important niche in software release management of component-based software.

The remainder of this paper is organized as follows. First, 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 tool. Based on those requirements, we describe the functionality of SRM, our software release management tool. This is followed by a description of some implementation details and a discussion of our experience in using SRM. We conclude with a discussion of related work and a look at some directions for future enhancements to SRM.

2. MOTIVATION

At first glance it may seem that adequate support for software release management can be found in configuration management tools [10–12], Web sites for software distribution [13–15], software deployment tools [16–18], or even some of the newer component platforms [4]. However, it is pertinent to observe that these solutions expose serious deficiencies and that no single effort has tackled the issues specifically or methodically. As a result, parts of the problem are addressed in different places, but an overall, cohesive, and comprehensive approach to software release management for component-based software is lacking. Consider the following examples.

- The GNU Free Software Directory [19] is a centralized index to most of the software developed under the GNU public license. Available software is described using a variety of textual metadata, including the components upon which a particular piece of software depends. Unfortunately, those dependencies are only listed by name, and locating and retrieving them is left to the user.
- Tucows [15] and ComponentSource [20] each provide an index to thousands of software components that are published and managed in a centralized manner on their respective Web sites. Compared to the GNU Free Software Directory, they provide the advantage of grouping components according to target platform. Unfortunately, however, they once again fall short in describing dependencies. Textual metadata lists dependencies by name and users are responsible for locating and retrieving the dependencies.
- The Red Hat package manager (RPM) [21,22] is one of the most advanced deployment systems available to date. A declarative file that contains the instructions for RPM to automatically install, configure, and uninstall the software accompanies each component managed by RPM. A particularly useful part of these instructions is the specification of dependencies. Each RPM package may contain a set of URLs pointing to other RPM packages on the Web. If necessary, RPM automatically downloads these packages and installs and configures the relevant components contained in those packages. Although powerful, this scheme leads to problems when a RPM package is removed or relocated from its original location. Other RPM packages dependent upon the component that was removed or relocated will no longer install properly.

Surprisingly, the situation in these three examples is better than in many others. Often one can find software for which different components need to be retrieved from different, geographically distributed *ftp* or Web sites. Moreover, some of these components may only be available in their latest versions, even if other components depend on older versions. The specification of dependencies is another issue, since such information may be spread out over source files, 'readme' documentation files, Web pages, and various other places. Typically, one still comes across situations in which the documentation of dependencies is incomplete, incorrect, or simply not present (see van der Hoek *et al.* [9] for a detailed description of one such example).

The kinds of problems we describe above are not unique to the GNU Free Software Directory, Tucows, ComponentSource, or RPM examples. Rather, they are fundamental and common to many instances of component-based 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, but all contributing to an overall product; and companies sometimes form virtual enterprises to create a joint product, assembled from sets of 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 ones 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, dependencies among components become increasingly complex.

From this discussion, it should be clear that software release management is an important part of the overall software process. Simply making available and retrieving interdependent components individually neither facilitates independent software development nor encourages widespread use of large component-based software. 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 in the context of distributed, decentralized, and autonomous component development.

3. REQUIREMENTS FOR SOFTWARE RELEASE MANAGEMENT

This section discusses a set of core requirements for software release management in the context of component-based software development. These requirements are derived mainly from our observations of a ten-year project in which four geographically distributed organizations collaborated in developing over fifty interrelated components. Over time, each of the components was released in multiple different versions. Each of these versions, in turn, could have different dependencies. Manual management of the resulting intricate web of dependencies turned out to be a difficult exercise and led to many problems in users obtaining a working software system [9].

The requirements address both developers and potential users of component-based software. For developers, a software release management process and support tool should provide a simple way to make component-based software available to potential users. This entails the following requirements.

- *Dependencies should be explicit and easily recorded.* It is critical for a developer to be able to easily and accurately document dependencies as part of the release process, even if those

dependencies cross organizational boundaries. Moreover, once recorded, those dependencies should be directly usable by the release management tool to automate parts of the process.

- *Releases should be kept consistent*. Specifically, actions by one organization should not render components by another organization useless. In particular, it should generally not be possible to remove components that are dependents of other components. Similarly, a newly released version of a component should not automatically replace an older version, since the two versions may in fact be incompatible.
- *The scope of a release should be controllable*. A developer should be able to control which users are able to download a component. As a minimum, licensing and access control mechanisms must be provided.
- *The release process should involve minimal effort on the part of the developer*. For example, when a new version of a component 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.
- *A history of retrievals should be kept*. This allows developers to track how often their components have been downloaded, as well as to feed this information into a separate contact database or e-mail list to be used for announcements of new releases, patches, related products, and the like.

For potential users (i.e. those organizations and individuals that obtain applicable components for some specific purpose such as experimentation, evaluation, or actual use), a software release management tool should provide an easy way to locate and retrieve such components. This leads to the following requirements.

- *Descriptive information should be available*. Based on this information, a user should be able to locate sets of components that may be of interest. Furthermore, a user should be able to determine whether a particular (version of a) component may be of use and, thus, should be downloaded for further examination.
- *Location transparency should be provided*. If desired, a user should be unaware of where components are physically stored.
- *A component and its dependencies should be retrievable as a single archive*. To avoid potential inconsistencies and mistakes when a user retrieves a component and its dependencies one-by-one from multiple different locations, a software release management system should allow a user to download a component and its dependencies in a single step and as a single archive.
- *Software deployment tools should be able to use the software release management tool as the source for components to be installed and configured*. In particular, the release management tool should, upon request, provide a deployment tool with metadata, dependencies, and the actual archives containing the components to be installed and configured.

A software release management process and support tool that satisfies these requirements will alleviate the problems evident in our motivating examples. They will make it easier for developers to release component-based software, and for users to efficiently and reliably obtain them.

4. A SOFTWARE RELEASE MANAGER

SRM is a software release management tool that we have designed, implemented, and used since 1996. The basics of SRM are similar to the traditional ‘label and archive’ paradigm used in the past. A SRM

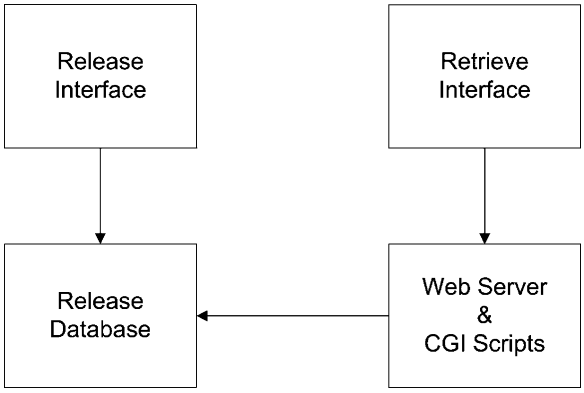

Figure 1. SRM architecture.

assumes that a configuration management system (e.g. ClearCase [10], Continuus [23], PVCS [12], or CVS [11]) is used to label, extract, and place a set of files into a release archive such as a *tar*, *zip*, or *jar* file. The exact content of an archive may be a set of source files, a binary file, an applet, or any other mechanism for capturing a component. The archive is subsequently published and made available to users, who in turn are responsible for locating and retrieving the archives in which they are interested. To make this process feasible for component-based software, SRM provides two important distinguishing features.

- *It structures the information used in the release management process*. SRM uses this structure in automating much of the process to support developers and users.
- *It provides location transparency*. In particular, dependencies can span multiple, physically distributed organizational boundaries. Both developers and users are capable of accessing the information regarding components in a way transparent to the location of the information.

Guided by the requirements discussed in Section 3, we arrived at a number of high-level design considerations that helped shape the overall architecture of SRM. First, we leverage existing HTTP protocols for communication within SRM. This allows users to retrieve components simply by using a standard Web browser. Second, we achieve location transparency by storing all artifacts in a physically distributed, but logically centralized database. This database integrally supports versioning of the artifacts. Finally, to reduce the effort of creating the Web pages through which components are released, as well as to provide some assurance that the information in those pages remains accurate with respect to the database, the Web pages are automatically generated. In addition to making the release of components easier for developers, this provides us with an opportunity to create a uniform interface through which all components can be retrieved.

Figure 1 illustrates the resulting architecture of SRM, which consists of four parts: a logically centralized, but physically distributed, release database; an interface through which developers place components into the release database; an interface through which users retrieve components from the

release database; and a Web server with *cgi* scripts through which users of the remote retrieve interface access the release database and download components. With the exception of the Web server and *cgi* scripts, which are straightforward, we discuss each part of the architecture in detail below.

4.1. The release database

The release database is a repository that SRM uses to store the metadata pertaining to the components, as well as the release archives themselves. The database is implemented using NUCM, a distributed repository for versioned artifacts [24]. SRM uses 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 in which multiple artifacts from multiple distributed sites are visible at the same time. It is physically distributed in that the artifacts are actually 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 usually stored in a repository that is physically present at that organization. Links are then put in the other physical repositories to allow each of the participating organizations access to the newly released component. Full details of this implementation, as optimized for flexibility and distributed operation, are beyond the scope of this paper, but can be found elsewhere [24].

Figure 2 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:

- release archives containing the actual components;
- metadata describing each component;
- a list of dependencies for each component; and
- Web pages for each component.

The release archives for the components and the 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. In the current version of SRM, the metadata and list of dependencies for each of the components are contained in a single file that is stored at just one of the sites. This happens to be site C in Figure 2. A more distributed schema has been explored for potential adoption in SRM [25].

4.2. The release interface

Through the release interface of SRM, developers *release* new components to the release database and *withdraw* (i.e. remove) obsolete components. Component releases are typically provided bottom up with respect to their dependencies, which is to say that before a component can be released, all other components upon which it depends must have been released. The inverse is true for withdrawing a release. To address the rare case of components with circular dependencies, as well as to support correcting erroneous dependencies and managing evolving dependencies, developers can *modify* a release through the release interface. Next, we discuss each of the release, modify, and withdraw parts of the release interface in more detail.

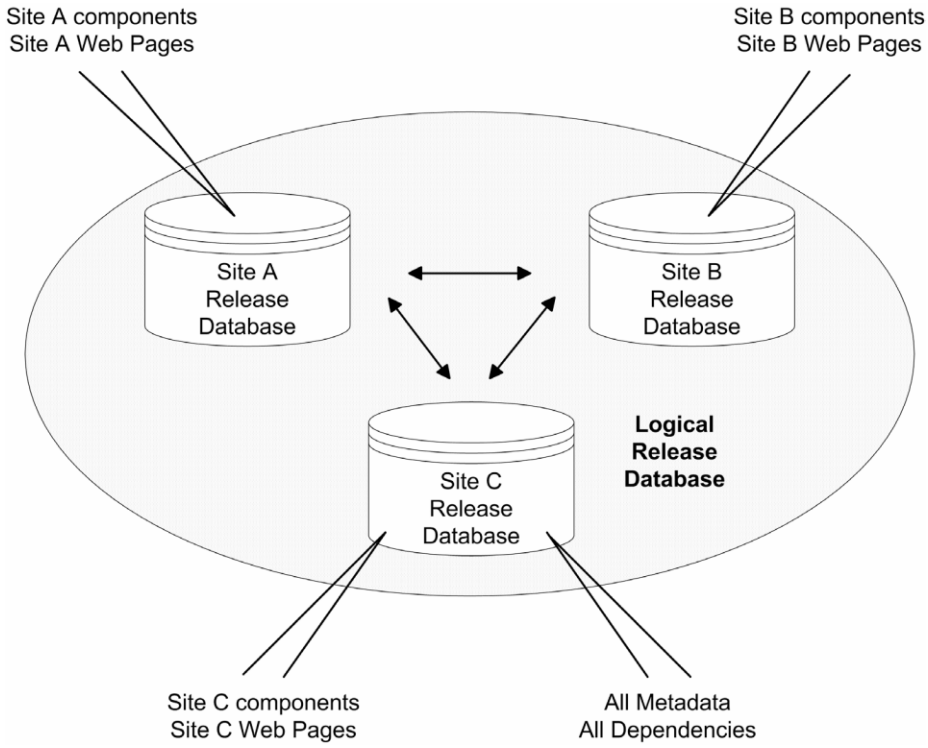

Figure 2. SRM release database.

Releasing a component

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

The minimal metadata that must be provided consists of a component name, a component version, and the name and e-mail address of a contact person. Additional metadata may be provided for descriptive purposes, such as, among others, the name of the developing organization, a Web site, a detailed description of the functionality of the component, and the platform(s) on which the component is known to properly install and execute. Although not required, developers are certainly encouraged to fill out all metadata, since it is the basis upon which potential users browsing the release database assess the suitability of a particular component for their purposes.

Metadata in SRM is descriptive. SRM does not perform any analyses or enforce any particular standards. This is a conscious design choice, since different organizations may be using different processes. Consider, for example, the version identifier field, which distinguishes different, co-existing

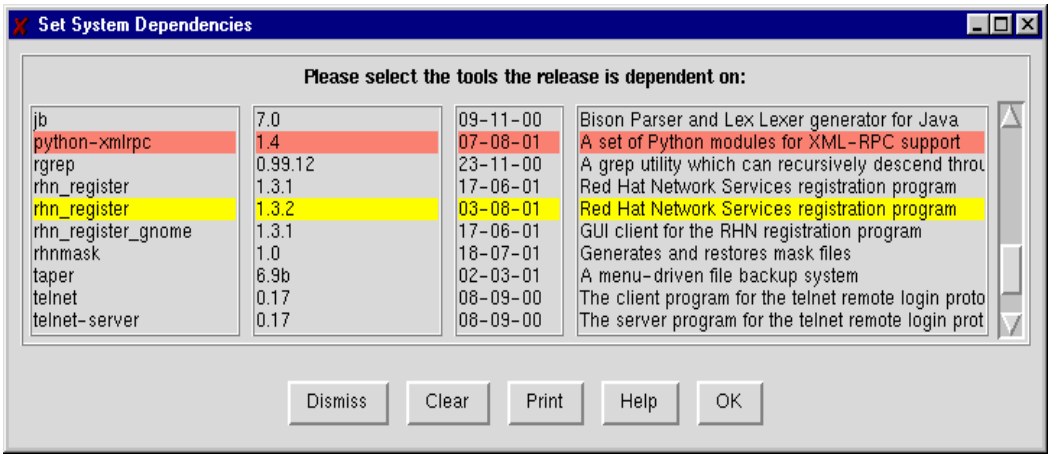

Figure 3. Selecting component dependencies.

releases of the same component within a release database. SRM could prescribe a particular versioning scheme or convention. Doing so, however, would be a mistake and actually decrease its applicability in the context of decentralized release management. Different organizations typically use different versioning schemes, and forcing all of those organizations to adopt a single versioning scheme just to use SRM is unreasonable. SRM, therefore, allows each organization to label its component releases according to its own versioning scheme.

To inform a potential user of the differences among component versions, one of the critical pieces of metadata that can be filled out is a textual description of the functionality that has changed between one component version and its predecessor. Users can peruse this field as it changes from release to release to determine the evolution of a component over time.

The second piece of information that a developer must provide describes the direct, first-level dependencies of a component on other components. SRM is able to calculate transitive dependencies across multiple components by following paths over first-level dependencies. Using a small subset of actual component releases that are currently available on the Red Hat Linux distribution Web site [26], Figure 3 shows an example of how SRM's user interface displays the specification of dependencies for a new component release, namely version `2.5` of the component `up2date`. (Note that the current version of the SRM interface uses the term 'tools' for dependent components and 'release' for the actual component being released.) The interface allows for a simple point-and-click selection of first-level dependencies. In this case, one dependency is selected and consequently highlighted by dark shading, namely `python-xmlrpc` version `1.4`. The SRM automatically includes a transitive dependency that it has calculated from previously recorded information regarding `python-xmlrpc` version `1.4`. In particular, lighter shading highlights `rhn_register` version `1.3.2` as a transitive dependency for the component `up2date` version `2.5`. The combination of the first-level and transitive dependencies is the complete set of dependencies maintained by SRM for any of the components being released.

It should be noted that specifying a remotely released component as a dependency is as easy as specifying a locally released component as a dependency, since the release database is transparently distributed. All released components, regardless of location, are uniformly shown in the dependency selection process.

It should also be noted that the information in the release interface is purposely brief. Developers generally are fully aware of which components they used during the development of their component. Selecting those components as dependencies is simply a matter of locating them quickly in the list of components presented. If for some reason more information is needed to reaffirm a particular choice, SRM can bring up more metadata regarding any of the listed components.

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

Finally, a developer must choose the scope of a component release. SRM provides three complementary mechanisms that can be used to control scope: release groups, an access control mechanism based on user names and passwords, and licenses.

Release groups are used to partition components into different tracks. For example, one could imagine dividing all the components that are available on the Red Hat Linux distribution site into five different groups: (1) kernel components; (2) applications; (3) patches; (4) experimental alpha releases; and (5) external contributions. In cases where thousands of components are released, partitioning is necessary to allow potential users to quickly find components that match their interest. Note that components can be part of multiple groups, allowing the partitioning of components along multiple dimensions and, thus, facilitating multiple paths of entry into the release database.

To control which potential users have access to the components in a release group, developers can specify a set of user names and associated passwords. This feature of SRM can be used, for example, to guarantee limited access to components that are released to in-house developers or paying customers. The first group would be protected with a username and password known only to the developers, the second with usernames and passwords that are distributed on a per-user basis whenever a component is purchased by an interested party.

Within a group, each component may have an associated license to which an interested user must agree before a component can be downloaded. The license text can be specified by the developer, and may range from a simple disclaimer to something as elaborate as the GNU General Public License [27]. In addition to a strictly enforced license, a developer may also choose to use a license in a manner similar to the traditional concept of a warranty. In such a case, the license is displayed after a component has been downloaded and only needs to be agreed to on a voluntary basis. This kind of license is generally used to present to an interested party a disclaimer or message that is not directly related to the download process itself.

SRM stores in the release database all metadata, all dependencies, and a copy of the archive containing each component. Old versions of a component are not removed and a 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 dependent components. In essence, SRM repository automatically documents the release management process.

One assumption we have made up to this point in the discussion is that all components are released through SRM. 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*

components. A developer who wants to release a component that is dependent on a foreign component simply releases two components. First, the developer releases the foreign component 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 a component stored inside SRM that can be used as a dependent component and retrieved by potential users. However, there is one important difference: for a foreign component, a developer specifies a URL through which that component can be retrieved, rather than providing an archive containing a release.

As an example, consider a developer who wishes to release a new chat component. The chat component uses `python-xmlrpc` version `1.4` as well as an external XML library provided by some other organization that does not use SRM. To release the chat component, the developer releases two components. First, the developer 'releases' the external XML library as a foreign component, providing to SRM its descriptive metadata and the URL through which it can be retrieved from its Web site. Second, the developer releases the chat component and selects, using normal point-and-click, `python-xmlrpc` version `1.4` and the library as dependencies. When a user attempts to download the chat system, SRM will use the given URL for the library to transparently retrieve the foreign component. It then returns it to the user as part of a single archive that also includes the chat system, the internally maintained dependency `python-xmlrpc` version `1.4`, and the automatically included transitive dependency `rhn_register` version `1.3.2`.

It should be noted that the use of URLs for foreign components breaks the consistency that a SRM otherwise guarantees, since a URL may become invalid over time. Nonetheless, the use of URLs forms a compromise in treating all dependencies in a uniform manner, irrespective of whether the dependencies are maintained by SRM. If absolute consistency must be guaranteed, a physical copy of the foreign component can always be stored inside the release database using normal release procedures.

The discussion thus far ignores the following three issues that are pertinent to software release management.

- *Variants*. At times, organizations release one particular version of a component as a series of related, but alternative variants. Consider, for example, an organization that releases different variants of its components for different customers or an organization that makes available both a lightweight, inexpensive variant of a component and a more function-rich, more costly variant. SRM provides two complementary mechanisms for dealing with variability. First, it is possible to use the version identifier to distinguish different variants. For example, one could release version `1.6-lightweight` and version `1.6-heavyweight` and use the rest of the metadata to further explain the difference. A second solution relies on the use of release groups. One could create a release group for each variant of a component (and its dependencies) and release different variants of the component to the appropriate release groups.
- *Platform dependencies*. SRM provides metadata entries through which developers specify both the software and hardware platform(s) upon which a component can be properly installed and executed. Because SRM is meant to be generic and cannot *a priori* predict all of the potential platforms upon which components may be installed (new languages, new versions of operating systems, and even new devices such as programmable PDAs continue to enter the marketplace), this entry is purely textual. SRM does not use these entries to, for example, determine the compatibility of components with target platforms. Nonetheless, it is often desirable to group

together those components that all operate on the same platform. Release groups can be effectively used for this purpose. By partitioning the complete set of available components in different release groups, one for each platform, users can assess whether or not the component in which they are interested is suitable for use in their organization. Because a single component release can be part of multiple release groups (e.g. Windows XP and Windows 2000), the release of components that operate on multiple platforms is supported. Moreover, it allows partitioning of components along multiple dimensions (e.g. a component can be part of both a Windows XP group and a JDK 1.4 group).

- *Patches*. The use of patches is a popular method of releasing component updates, and is often used to make available bug fixes. In contrast to releasing a whole new version of a component, a patch only contains a small set of changes to be applied to a specific version of a component. SRM naturally supports the use of patches. A developer simply releases a patch and specifies as a dependency the particular component version upon which the patch is built. When patches are released incrementally, the dependencies form a chain of patches, the last of which is dependent on a particular component version. The advantage of such a chain is that a user is not only guaranteed to retrieve the right set of patches, but is also informed of the right order in which to apply them.

Note that these techniques work irrespective of whether a component is directly released to SRM or wrapped as a foreign component. Proper use of metadata and release groups uniformly supports variants, platform dependencies, and patches in both cases.

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 or e-mail address, may change. A more important reason is that underlying dependencies may change. For example, `python-xmlrpc` version `1.4`, as shown in Figure 3, depends on `rhn_register` version `1.3.2`. Because the latter component evolves independently of the first one, it may happen that a new version is created (for example, to fix a bug or to add new functionality) while `python-xmlrpc` stays the same. If the changes to `rhn_register` leave its interface and behavior intact, `python-xmlrpc` will not need to change to interoperate with the new version. Once it is determined that `python-xmlrpc` version `1.4` works properly with the new version of `rhn_register`, SRM can be used to switch the version of the dependent component `python-xmlrpc` version `1.4` to the new version. Notice that a mechanism based on a default version, such as 'the latest', would not work in this scenario. This is because the version of the dependent component would have switched automatically from the old to the new before it was verified that the new version was compatible.

Changes in dependencies are not automatically propagated to users, since that would be a violation of their autonomy. Rather, users have to decide when they are ready to upgrade, download the necessary components, and most likely invoke a deployment system to properly install and configure the newly downloaded components. SRM keeps a history of retrievals and allows users to register for future notifications (see Section 4.3), which helps in creating a list of users to be contacted when dependencies change.

The need to change the scope of a release is more common than the need to modify dependencies without creating a new version. To change the scope, developers can change the release groups to which a component belongs, modify the access control lists that govern access to the components in a release group, and adjust the enforcement and text of licenses as necessary.

Withdrawing a release

The third operation supported by the release interface allows developers to withdraw components from the release database. This functionality is provided to make an obsolete or non-supported component unavailable for download and delete it altogether from the release database. To guarantee the integrity of the release database, one restriction is placed on the withdrawal of components: they can only be withdrawn if they are not serving as a dependency for other components. For example, since `python-xmlrpc` version `1.4` depends on `rhn_register` version `1.3.2`, `rhn_register` version `1.3.2` cannot be withdrawn from the release database. If a developer still wants to withdraw `rhn_register` version `1.3.2`, either `python-xmlrpc` version `1.4` needs to be withdrawn first, or the dependencies of `python-xmlrpc` version `1.4` need to be changed to include a different version of `rhn_register`.

A second way of withdrawing a component is to change its scope by removing it from one or more release groups. The component will then no longer be visible to a potential user who browses the components in those release groups, but is still present in the database for use as a dependency. This mechanism allows fine-grained control over the audiences to which a component will still be visible, thereby making it possible to phase out a component in a controlled manner. Once a component has been removed from all of its release groups, primary downloads of the component will be radically reduced, but the component can still be downloaded as part of another component release until it is explicitly deleted from the release database.

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's deployment environment.

The retrieve interface is built as an automatically generated, static, Web-page-based interface to allow widespread access to the release database. Every time a component is released, SRM creates a Web page for that component and updates the Web pages for the release groups to which the component belongs. These release-group Web pages are the primary access points to the release database and each presents a listing containing short synopses of the components that are part of the group.

Upon selection of a component from a group Web page, the Web page corresponding to the selected component is presented to the user. This Web page contains all of the descriptive metadata supplied by the developer, allowing a user to assess the suitability of the selected component. In addition, the Web page shows the dependence graph for the component and provides selection buttons to manually turn off or on dependencies for retrieval. This portion of the interface is shown in Figure 4. In this example, `up2date` version `2.5` is being retrieved, and both of its dependencies have been selected for retrieval as well.

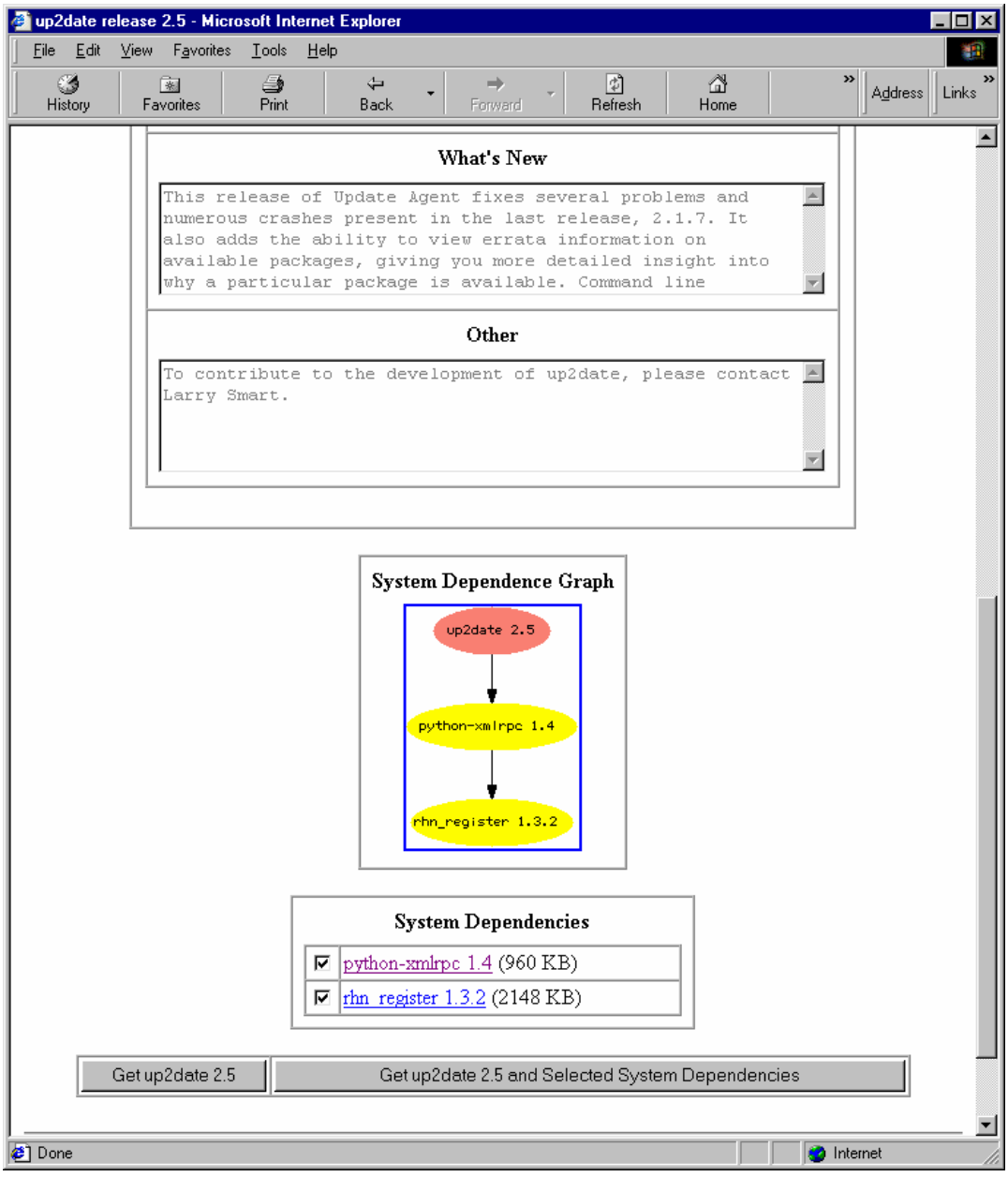

Figure 4. Downloading a component using the SRM retrieve interface.

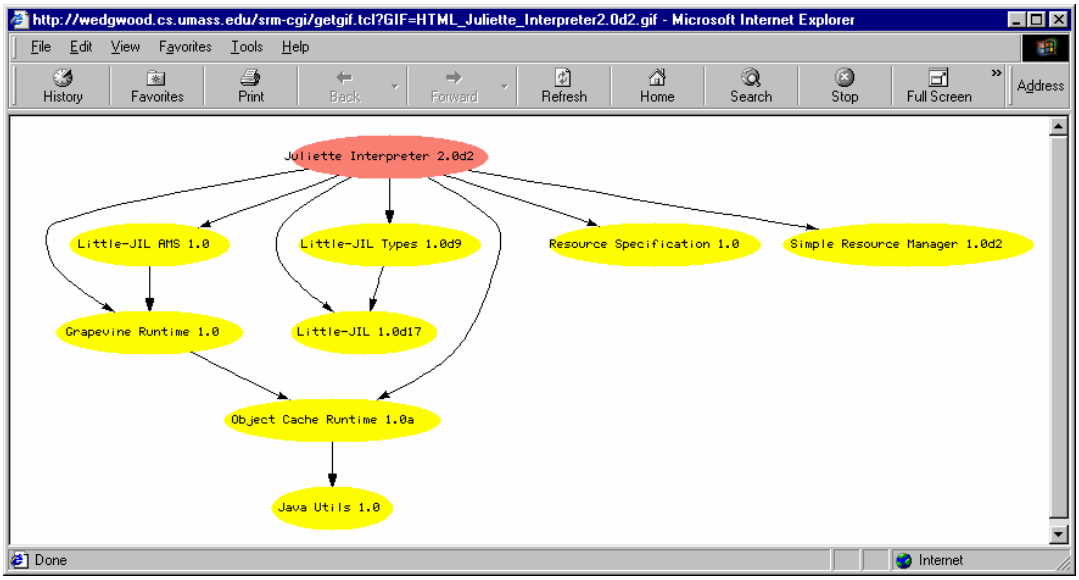

Figure 5. Actual dependence graph showing a richer set of dependencies.

Upon confirmation of the selection, SRM assembles an archive containing all selected components and delivers this archive to the user. In case SRM delivers a single component, the format of the archive maintains that of the original *tar* file, *zip* file, or *jar* file. If, on the other hand, multiple components are delivered, the current implementation of SRM assembles the individual archives into a single *tar* file. Upon receipt, a user first must *untar* this archive to gain access to the archives containing the individual components. In today's Web browsers, such functionality is usually automatic.

In case of richer dependencies than those shown in the example of Figure 4, it is possible that a single component is listed as a dependency for multiple other components. Because of SRM's detailed tracking of dependencies, it is able to determine such occurrences and, consequently, will only include a single instance of the component in the archive that is delivered to the user. For example, consider the dependence graph shown in Figure 5. This graph was obtained from an actual installation of SRM at one of its user sites. As the graph shows, two components specify the component `Object Cache Runtime` version `1.0a` as a dependency. Since SRM is aware of this fact, it will only include a single copy of this component upon download of the component `Juliette Interpreter` version `2.0d2` with all of its dependencies. The same holds for the component `Grapevine Runtime` version `1.0`.

Cyclic dependencies are managed in a similar manner. SRM detects their presence and presents users with an archive in which each component that is part of the cycle is included only once. SRM takes versions into account when determining the presence or absence of cycles. Different versions of the same component are allowed in a single archive and do not trigger the detection of a cycle. This

facilitates situations in which parts of a software system use an old version of a component and other parts have already been upgraded to a newer version. While such situations may lead to installation or configuration problems, there are cases in which it is legitimate and necessary to include multiple versions of a single component in a single archive, hence the support by SRM.

A user of the retrieve interface is not aware that the various components might have originated from several geographically dispersed sites. Physical distribution is hidden within SRM release database, which silently retrieves the components from the various repositories and ships all of them back to the user as a single archive. From there, the user is responsible for further installation. The goal of SRM, after all, is only to deliver the right set of components to a user. Nevertheless, a specialized software deployment tool may subsequently be used to install the components that have been retrieved. An experimental integration of SRM with one such system, the Software Dock [28], was made to demonstrate the feasibility of this approach.

Before giving a user access to the archive containing the desired components, SRM verifies whether one or more of the components in the archive have an associated license. If so, it displays these licenses first and requires a user to agree with their contents before access to the archive is granted. Developers using SRM, thus, can be assured of a process in which their licenses are explicitly presented and read.

After a user has downloaded the desired components, the SRM retrieve interface optionally presents them with a registration page. On this page, the user can fill out their name, e-mail address, and reason for downloading the selected components. This information is forwarded to the developers of the downloaded components, allowing them to maintain a list of users who may be interested in receiving product updates.

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

5. IMPLEMENTATION AND EXPERIENCE

SRM was first developed in late 1996 to demonstrate the utility of the NUCM repository for distributed configuration management [24] in addressing a real-world software release management problem concerning four universities jointly developing and releasing an interrelated set of about 50 components [9]. SRM has since undergone a series of extensive revisions, culminating in the system that is available today. NUCM still serves as the release database. The release interface is implemented in Tcl/Tk [29], whereas the retrieve interface is implemented as a series of statically generated Web pages that are accessed from a standard Web browser via *cgi* scripts [30].

In total, SRM comprises about 18 000 lines of source code, approximately 80% of which is dedicated to presenting an appropriate user interface for the release process and generating Web pages for the retrieve process. The remaining 20% implement: (1) the storage and retrieval of component releases in the release database; (2) the *cgi* scripts that return requested metadata, component releases, and licenses to a user's Web browser; and (3) the storage of download statistics and user registrations in the release database. All parts of SRM are designed to handle concurrency by using NUCM's locking primitives in implementing a two-phase commit protocol. As a result, multiple users can simultaneously download sets of interrelated components, even as multiple developers release, modify, and remove components at the very same time.

SRM has been successfully installed and tested on a variety of Unix platforms (e.g. Solaris, HP-UX, and Linux). The limitation to Unix platforms is largely due to the use of NUCM as the release database.

SRM itself could easily be ported to non-Unix platforms. However, such an effort would require migrating to a different release database.

SRM has been downloaded over 600 times over the four-year period of 1997 to 2001. We ourselves have used it extensively, which has allowed us to evaluate its user interface and basic functionality first hand. In fact, many modifications and enhancements were made over the years, resulting in the system presented here. Of the users who downloaded SRM, about 150 registered to receive update announcements, indicating the usefulness of this part of the retrieve interface in building a community of users and allowing users the freedom to decide whether or not to be part of this community.

Our primary use of SRM has been as the release management system for the software produced in the Software Engineering Research Laboratory (SERL) at the University of Colorado. It has been used to release about ten different systems in a number of different versions. Over 1800 downloads were performed using the SERL SRM site. The benefits for the developers of the components released by SERL are evident. First, SRM informs them of the number of downloads, in essence providing a 'popularity' gauge. Second, very little time needs to be spent on creating Web pages and recording dependencies to release a new version of a component, since SRM does so automatically and only needs as input the metadata and dependencies that have changed as compared to the previous version of the component. Lastly, component releases are consistent. A user who downloads a set of components with their dependencies should not experience any installation or execution problems that are due to incompatible versions (unless, of course, the wrong information was provided by the developers).

The inherently distributed nature of SRM was tested during an 18-month tenure as the software release management tool for the Evolutionary Design of Complex Software (EDCS) program sponsored by the United States Defense Advanced Research Project Agency (DARPA). Participating organizations from all over the U.S. released about 15 components to SRM, which were retrieved about 150 times. Due to the specific domain and experimental nature of the components, the usage numbers of the EDCS release database were not as high as one might expect. Nonetheless, this particular installation demonstrated an important aspect of SRM in being fully capable of supporting a group of autonomous, loosely collaborating organizations that all resided in physically different locations. For the EDCS program, SRM created a single resource where components were released perused, assessed, and retrieved, thereby reinforcing the critical role software release management plays in such an environment.

6. RELATED WORK

In this section, we discuss the four areas of work most closely related to SRM. First, we examine relevant Web sites through which software is distributed. Second, we present an overview of software deployment, which is the larger context of software distribution, installation, and configuration in which software release management plays a crucial role. Third, we look at the field of configuration management. Finally, we discuss component platforms.

6.1. Web sites for software distribution

Web sites have long been used to distribute software [25]. Initially, they were used in an *ad hoc* manner to distribute software that an individual person or organization had produced. The ability to link to different locations was soon after explored to create Web sites that served as indexes to available

software. The rapid increase in popularity of these indexes, combined with the business opportunity to generate revenue out of advertising, quickly led to the development of professionally maintained Web sites that provide the ability to not only search for and download appropriate software, but to also upload software to be published. Exemplified by Gamelan [13] and Tucows [15], these Web sites now index and make available thousands of software titles and are remarkably popular.

The influence of component-based software can be found in several smaller-scale, and usually more focused, indexes. The open source domain [31] provides a particularly useful example: since the trend is to build upon components provided by other developers, open source Web indexes such as freshmeat [32] and SourceForge [14] have started to keep track of dependencies among components. Specifically, the description of each available component contains an explicit section with a list of other components that a user must obtain before the desired component can be installed, configured, and run.

Two important differences exist between these Web sites and SRM. First SRM is specifically designed to provide management of component releases and their dependencies in a distributed and decentralized setting. Most Web sites, on the other hand, require all component releases to be managed at their central site. The second difference lies in the use of dependencies to support the download process. Whereas the Web sites provide a rudimentary mechanism that lists necessary components but requires users to download them one-by-one, SRM automatically bundles a component and its dependent components, and delivers them at once in a single archive.

6.2. Software deployment

Software release management is an important part of the overall process of software deployment. Defined by the software deployment life cycle [33], software deployment covers those activities that occur after a piece of software has been developed. Specifically, software deployment covers the release and retirement of software by its producer; the installation and removal of software on a particular user machine; the update, reconfiguration, and adaptation of installed software to change its desired behavior; and the activation and deactivation of installed software to ensure proper initialization and destruction sequences are adhered to when the software is executed. One of the earliest software deployment systems was AT&T's ship [16], which since has been followed by a host of commercial systems (e.g. InstallShield [17], NetDeploy [18], Tivoli [34,35]), an open source system (RPM [21,22]), and a comprehensive research system (Software Dock [33]).

The influence of component-based software is as much visible in software deployment as it is in Web sites for software distribution. The Software Dock and RPM both were specifically designed to manage dependencies among components. For example, both prohibit the removal of installed components upon which other installed components rely for their proper functioning. Similarly, InstallShield and NetDeploy are able to check, prior to installation of a component, whether all of the other components are already installed for the component itself to be installed and function. The Software Dock and RPM take this even further and will automatically download and install necessary components.

The primary difference between these deployment systems and SRM lies in their focus. Although deployment systems do incorporate some rudimentary form of software release management, their focus on installation and configuration typically limits their support to the retrieval of information and components through a simple form of connectivity between the deployment tool and the original release site. This, as can be seen in Section 3, is only a small portion of the functionality needed to fully support software release management.

6.3. Configuration management

Configuration management systems (e.g. ClearCase [36], Continuus [23], CVS [37]) have long been used to manage the evolution of software. In precisely capturing different versions of a software system and its constituent source files, the use of a configuration management system creates a historical record of all changes made during the development process. Baselines [38] are used to label and record important sets of versions of artifacts, such as those used in a particular release.

The influence of component-based software development on configuration management can be found in the advent of vendor code management [39]. Vendor code management allows an organization to periodically import into its configuration management system source code developed by another organization using another configuration management system. This allows the organization to incorporate and assume control over 'foreign' components in the development of its software. A similar mechanism is used in source tree composition [40], a technique that refines vendor code management to support public domain software.

Configuration management systems and SRM serve a complementary role. Whereas configuration management systems focus on managing the development and evolution of software components, SRM supports the actual release and download of archives containing such components. No current configuration management system supports the kinds of functionality that SRM provides.

6.4. Component platforms

Recent advances in component technology present some interesting alternative views on the issue of release management and dependencies. For instance, component models such as .NET [4] inherently support the specification of dependencies, which are automatically downloaded by a supporting infrastructure to instantiate a component at run time. Each component has a version identifier, which adheres to a specific scheme to encode whether or not the interface of a component is backward compatible from version to version. While at first .NET may seem to make SRM obsolete, the truth is that SRM and .NET are complementary: the .NET mechanism of specifying dependencies as URLs still needs a reliable release archive that guarantees the consistency and presence of those dependencies over time. URLs by themselves cannot do so, but using the URLs to mimic those used by SRM retrieve interface can provide this functionality. Doing so is trivial, since the SRM retrieve interface already relies on the standard HTTP protocol to retrieve desired components.

The system that is most similar to SRM is the Aonix Select Component Manager [41], which provides organizations with a comprehensive component repository. The component repository maintains dependencies, versions components, publishes and catalogues components, and supports searching and distribution of components. A strongpoint of the Aonix Select Component Manager is its integral use of structured metadata in not only a descriptive manner, but also as the basis for consistency analyses. While clearly stronger than SRM in this manner, SRM is unique in its support for managing dependencies across multiple, distributed, and decentralized release sites.

7. CONCLUSIONS

The work described here represents a novel approach to the software release process for component-based software. By means of a software release management tool, a bridge has been built between the

development process and the deployment process. 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, supports the release of components by autonomous and decentralized organizations, promotes the explicit use of dependencies, and supports users in obtaining a consistent set of components.

The basic concept behind SRM is simple: to provide distribution transparency to the release of software components while guaranteeing consistency of the dependencies among those components. However, its fundamental contribution of explicitly recognizing and supporting release management is extremely important. In the past, this part of the software development process was largely ignored. Its essential role in component-based software development, however, necessitates a system like SRM to explicitly support the process that takes place.

SRM has another advantage that extends beyond the scope of component-based software. In settings where a set of semi-independent groups is cooperating to build one or more software products, 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 groups. Each of the groups is still able to use its own development process, its own configuration management system, and its own development tools. However, the various mechanisms for release management are 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. In particular, SRM provides developers with 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, it becomes the language and mechanism for intergroup communication about interdependent components.

We believe SRM represents only the beginning of a research direction that further elaborates and supports software release management as a distinct role between development and deployment of component-based software. At the forefront of our future research are the following two issues that we deem most important in advancing software release management.

- *Exploring further interoperability with configuration management and software deployment systems*. We already performed a simple integration with an existing software deployment system [28]. Our experience with this integration illustrates that many issues remain to be addressed. For example, we want to automatically import dependencies from a configuration management system, rather than making developers specify them by hand (consider having to document one-by-one all the dependencies of a COM [42] component). In addition, we believe SRM needs to evolve into a bi-directional bridge between the domains of configuration management and software deployment. To truly gain control of component-based software development, it is necessary to deploy and install components into a configuration management system and automatically release components out of a configuration management system [43].
- *Exploring the relationship of SRM to component models*. Recent component models such as .NET [4] allow the specification of dependent components as structured metadata inside a component itself. Upon instantiation of the component, its dependent components are downloaded and instantiated as well. Unfortunately, in .NET dependencies are specified as URLs, which may change or disappear just as in the RPM example of Section 2. A separate release management tool that circumvents these kinds of problems is still required. We believe

that an integration of SRM with a tool that resolves dependencies at run time can guarantee that a consistent set of components is delivered and executed.

The implementation of SRM as described in this paper can also be improved. The available technology has significantly advanced since SRM was first developed, and we believe a more dynamic, non-HTML version of SRM will provide users with a much friendlier experience. Finally, we realize that the problem of managing the release of distributed, interdependent components can be generalized to other domains. As such, we have begun experimentation with a generic version of SRM, which inherits all of the functionality described in this paper, but is based on an XML-based engine to allow full configuration of the metadata, distribution mechanism, and user interface [25].

ACKNOWLEDGEMENTS

We would like to thank the present members and past graduates of SERL for their invaluable contributions that led to the current incarnation of SRM. In particular, we wish to acknowledge Robert Smith and Michael Hollis for their contributions to the design and implementation of SRM, and Dennis Heimbigner, Antonio Carzaniga, and Richard Hall for their extensive trial use and continuous feedback. We also wish to thank the participants in the DARPA EDCS program for their willing use of SRM.

This effort is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement Nos. F30602-00-2-0599 and F30602-00-2-0608. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

REFERENCES

1. Szyperski C. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press: Boston, MA, 1998.
2. Crnkovic I *et al*. *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering: Component Certification and System Predication*. Software Engineering Institute: Pittsburgh, PA, 2001.
3. Heineman GT, Councill WT (eds.). *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley: Reading, MA, 2001.
4. Platt DS. *Introducing Microsoft Dot-Net*. Microsoft: Redmond, WA, 2001.
5. Matena V, Hapner M. Enterprise Java Beans Specification, v1.1. ftp://ftp.java.sun.com/pub/ejb/11final-129822/ejb1_1-spec.pdf [2000].
6. van Ommering R *et al*. The Koala component model for consumer electronics software. *Computer* 2000; **33**(3):78–85.
7. Aldrich J, Chambers C, Notkin D. ArchJava: Connecting software architecture to implementation. *Proceedings Twenty-fourth International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos, CA, 2002; 187–197.
8. Sreedhar VC. Mixin'Up components. *Proceedings 24th International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos, CA, 2002; 198–207.
9. van der Hoek A *et al*. Software release management. *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Springer: Heidelberg, Germany, 1997; 159–175.
10. Allen L *et al*. ClearCase MultiSite: Supporting geographically-distributed software development. *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*. Springer: Heidelberg, Germany, 1995; 194–214.
11. Fogel K. *Open Source Development with CVS*. Coriolis: Scottsdale, 1999.
12. Merant. *Using PVCS for Enterprise Distributed Development*. Merant: Hillsboro, OR, 1998.

13. Earthweb. http://www.gamelan.com/ [2001].
14. SourceForge. http://www.sourceforge.net/ [2001].
15. Tucows. http://www.tucows.com/ [2001].
16. Fowler G *et al*. Libraries and file system architecture. *Practical Reusable UNIX Software*, Krishnamurthy B (ed.). Wiley: New York, NY, 1995; 78–90.
17. InstallShield. http://www.installshield.com/ [2001].
18. NetDeploy. http://www.netdeploy.com/ [2001].
19. The GNU Foundation. http://www.gnu.org/directory/index.html [2001].
20. ComponentSource. http://www.componentsource.com/ [2002].
21. Bailey EC. Maximum RPM. http://www.rpm.org/max-rpm/ [2002].
22. Red Hat. http://www.rpm.org/ [2001].
23. Continuus Software Corporation. *Distributed Code Management for Team Engineering*, 1998.
24. van der Hoek A *et al*. A testbed for configuration management policy programming. *IEEE Transactions on Software Engineering* 2002; **28**(1):79–99.
25. Smith RA. Analysis and design for a next generation software release management system. *MS Thesis*, University of Colorado at Boulder, 1999.
26. Red Hat. http://www.redhat.com/products/software/linux/ [2001].
27. The GNU Foundation. http://www.gnu.org/licenses/gpl.html [2001].
28. Hall RS *et al*. An architecture for post-development configuration management in a wide-area network. *Proceedings 1997 International Conference on Distributed Computing Systems*. IEEE Computer Society: Baltimore, MD, 1997; 269–278.
29. Ousterhout JK. *Tcl and the Tk Toolkit*. Addison-Wesley: Reading, MA, 1994.
30. Guelich S, Gundavaram S, Birznieks G. *CGI Programming with Perl* (2nd edn). O'Reilly: Cambridge, MA, 2000.
31. Raymond ES. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly: Cambridge, MA, 2001.
32. Freshmeat. http://www.freshmeat.com/ [2001].
33. Hall RS, Heimbigner DM, Wolf AL. A cooperative approach to support software deployment using the software dock. *Proceedings of the 1999 International Conference on Software Engineering*. ACM Press: New York, NY, 1999; 174–183.
34. Tivoli Systems. *Application Management Specification*. Tivoli: Austin, TX, 1997.
35. Tivoli Systems. http://www.tivoli.com/ [2001].
36. Atria Software. *ClearCase Concepts Manual*. Atria Software: Lexington, MA, 1992.
37. Berliner B. CVS II: Parallelizing software development. *Proceedings of 1990 Winter USENIX Conference*, Washington, DC. Usenix Association: Monterey, CA, 1990.
38. Conradi R, Westfechtel B. Version models for software configuration management. *ACM Computing Surveys* 1998; **30**(2):232–282.
39. Continuus Software Corporation. *Continuus Task Reference*, 1994.
40. de Jonge M. Source tree composition. *Technical Report SEN-R0204*, CWI Amsterdam, The Netherlands, 2001.
41. Aonix. Aonix Select Component Manager. http://www.aonix.com/content/products/select/select_compman.html [2002].
42. Sessions R. *COM and DCOM: Microsoft's Vision for Distributed Objects*. Wiley: New York, 1997.
43. van der Hoek A. Integrating configuration management and software deployment. *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*. DSTC: Brisbane, Australia, 2001.

