

# Managing the Evolution of Distributed and Inter-related Components

Sundararajan Sowrirajan and André van der Hoek

School of Information and Computer Science  
University of California, Irvine  
Irvine, CA 92697-3425 USA  
{ssowrira, andre}@ics.uci.edu

**Abstract.** Software systems are increasingly being built by integrating pre-existing components developed by different, geographically distributed organizations. Each component typically evolves independently over time, not only in terms of its functionality, but also in terms of its exposed interfaces and dependencies on other components. Given that those other components may also evolve, creating an application by assembling sets of components typically involves managing a complex web of evolving dependencies. Traditional configuration management systems assume a form of centralized control that simply does not suffice in these situations. Needed are new configuration management systems that span multiple organizations, operate in a distributed and decentralized fashion, and help in managing the consistent evolution of independently developed, inter-related sets of components. A critical aspect of these new configuration management systems is that they must respect the different levels of autonomy, privacy, and trust that exist among different organizations. In this paper, we introduce TWICS, an early example of such a new configuration management system. Key aspects of TWICS are that it maintains traditional configuration management functionality to support the development of individual components, but integrates policy-driven deployment functionality to support different organizations in evolving their inter-related components.

## 1 Introduction

Component-based software development has perhaps received the most attention of any kind of software development method to date [13,21]. Based on the premise of component reuse, the hypothesis is that application assembly out of pre-existing components brings with it significant savings in terms of time and cost [26].

While much effort has been put towards developing component technologies via which individual components can be implemented and subsequently assembled into applications (e.g., .NET [18], EJB [7], CORBA [17]), tool support for component-based software development has lagged behind. Exceptions exist (e.g., design notations such as UML [20], component repositories such as ComponentSource [4], and

composition environments such as Component Workbench [16]), but more often than not it is still assumed that existing tools suffice in their present incarnations.

Configuration management systems suffer from this misperception. Although well equipped to support the needs of traditional software development, it has been recognized that they do not meet the unique demands of component-based software development [12,29]. The assumption of centralized control underlying current configuration management systems is at the core of this problem. Since different components typically are developed by different organizations in different geographical locations, centralized control cannot be enforced.

Needed is a new kind of configuration management system that explicitly supports component-based software development. Such a configuration management system must span multiple organizations, operate in a distributed and decentralized fashion, and help in managing the consistent evolution of independently developed, inter-related sets of components. For a component producer, this means that the configuration management system must support development and publication of components. For a component consumer, this means that the configuration management system must support the organization in obtaining components from other organizations, controlling the local evolution of the obtained components independent from their remote schedules of new releases, and publishing its own components with appropriate documentation of their dependencies. In essence, the configuration management system must integrate traditional configuration management functionality with what is known as software deployment functionality: automatically publishing, releasing, and obtaining components and their dependencies [10].

In this paper, we introduce TWICS (Two-Way Integrated Configuration management and Software deployment), an early example of a configuration management system specifically designed to address component-based software development. TWICS is built as a layer on top of Subversion, an existing open source configuration management system that supports the traditional code development paradigm [24]. TWICS adds a component-oriented interface and wraps all the Subversion commands to operate on components as a whole rather than individual source files. Moreover, TWICS adds deployment functionality in the form of an interface through which the release of components and their dependencies can be controlled, and through which remote components can be downloaded and placed in the local repository. Finally, TWICS respects the different levels of autonomy, privacy, and trust that may exist among different organizations. To do so, it allows individual organizations to establish and enforce policies that govern such issues as to whether a component can be obtained in binary or source form and whether a component may be redistributed by other organizations.

The remainder of the paper is organized as follows. In Section 2 we discuss a typical component-based software development scenario and illustrate the difficult issues that it poses for a traditional configuration management system. Section 3 provides a brief background on the disciplines of configuration management and software deployment, which form the basis for our overall approach described in Section 4. Then, in Section 5, we describe the detailed capabilities of TWICS. We conclude with an outlook at our future work in Section 6.

## 2 Example Scenario

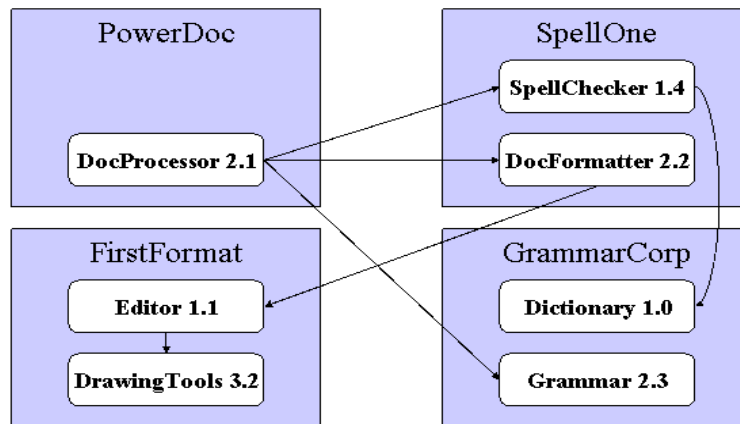
To exemplify the issues raised by component-based development for a traditional configuration management system, consider the scenario depicted in Figure 1. Grey boxes represent different organizations, ovals components developed by each organization, and arrows dependencies among the components. The organization SPELLONE, for example, develops two different components: SPELLCHECKER and DOCFORMATTER. The component SPELLCHECKER depends on the component DICTIONARY as developed by the organization GRAMMAR. The component DICTIONARY has no further dependencies, but the component SPELLCHECKER is used by the component DOC-PROCESSOR as developed by the organization POWERDOC. Many more component dependencies exist, creating an intricate web of distributed, inter-dependent components. Note the use of version identifiers: since each individual component evolves independently from the other components, dependencies must be on a per-version basis.

A first challenge arises because some components may be available solely in binary format, some in source code format, and others maybe even in both. In the case of source code, an organization that obtains such a component may make some local modifications to the component. As newer versions of the component become available, the local changes must be carefully combined with the changes by the original developer in order not to lose any functionality. For example, POWERDOC may be using a locally modified version of GRAMMAR version 2.3. As GRAMMARCORP puts out a new version of the component, version 2.4, POWERDOC must obtain that new version, bring it into the local configuration management system, and ensure that its local changes are integrated in the new version.

A second challenge arises when an organization decides to release its component. At that point, its dependencies on other components must be precisely documented such that anyone obtaining the component knows which other additional components it must obtain. In some cases, an organization may simply document the dependencies. In other cases, it may include the dependencies as part of its release. In yet other cases, a mixed approach may be followed. In all cases, however, it is critical that the dependencies, which may be both direct and indirect, are precisely documented. In the case of POWERDOC, it has to include all of the dependencies of DOCPROCESSOR 2.1, which indeed spans all components by all four organizations in our example. Clearly, this is a non-trivial and error-prone task that often has to be performed manually.

A third challenge arises when organizations have different trust policies between them. A component producer can choose to provide different types of access permissions (source code access or binary access) and redistribution permissions for its components to different component consumers. For example, GRAMMARCORP might be willing to share the source code of its DICTIONARY version 1.0 component with SPELLONE, but may be willing to provide only the binary of GRAMMAR version 2.3 component to POWERDOC. On the other hand, GRAMMARCORP may be willing to allow redistribution of GRAMMAR version 2.3 by POWERDOC, but may not provide similar privileges to SPELLONE for DICTIONARY VERSION 1.0. Clearly, there should be a means for expressing such trust policies between companies and such policies

must be incorporated as part of the component's deployment package to ensure they are adhered to.



**Figure 1.** Example Scenario of Component-Based Software Development.

The above challenges are summarized by the following four questions:

- How can components that are developed by external organizations be efficiently incorporated into the local development environment?
- How can the remote evolution of those components and their influence on the locally developed software systems be controlled?
- How can a component-based software system be effectively deployed as a consistent set of inter-related components?
- How can the issues of autonomy, privacy, and trust of the various organizations be preserved throughout this process?

While it is clear that the availability of a separate configuration management system and a separate software deployment system help in addressing these questions, we observe that a single system that integrates configuration management and software deployment functionality would be more desirable. Such a system, for example, can automatically place deployment information regarding which components were obtained from which other organizations in the local configuration management system. That information can then be used to support the release of a component that depends

on those components. TWICS is such a system, and is specifically designed to support scenarios like these.

### **3 Background**

TWICS builds upon work of two areas, namely configuration management and software deployment. We discuss relevant contributions in each of these areas below.

#### **3.1 Configuration Management**

Many configuration management systems have been built to date [3,5]. All share the same basic goal: to manage the evolution of artifacts as they are produced in the software life cycle. Most current configuration management systems still follow the model introduced by early systems such as RCS [23] and CVS [2]. Artifacts are locally stored in a repository, can be accessed and modified in a workspace, and, if modified, are stored back in the repository as a new version. To manage concurrent changes, either a pessimistic approach (locking) or an optimistic approach (merging [14]) is used.

Two particular shortcomings of current configuration management systems make it difficult for these systems to support configuration management for component-based software. First, current configuration management systems are very much focused on managing files. While system models have been devised that operate at a higher-level of abstraction [8,9,27], principled use of these system models requires users to manually map them onto source code.

The second shortcoming lies in the fact that current configuration management systems assume centralized control regarding the evolution of artifacts. Only two systems provide support for external components. The first, CM/Synergy [22], provides vendor code management for automatically incorporating new versions of source code from an external source. Similarly, auto bundle [6] supports the incorporation of external components in a local configuration management system and even supports packaging of a resulting set of components.

Both systems are clearly a step in the right direction with respect to the problem of managing component-based software development. However, CM/Synergy only solves one aspect of the problem, namely the incorporation of other components into the local environment. Subsequent deployment is not supported. Similarly, auto bundle assumes that only source code is shared among different organizations. Neither system addresses the different needs of autonomy, privacy, and trust as required by different collaboration policies.

### 3.2 Software Deployment

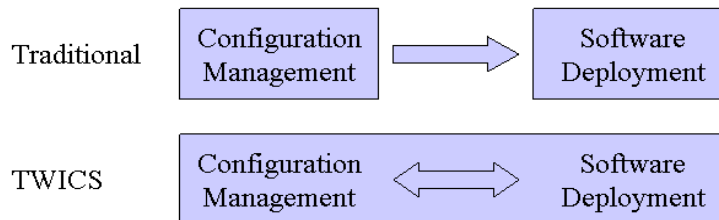
Software deployment is concerned with the post-development phase of a software product's lifecycle [10]. In particular, a software deployment system typically supports the activities of packaging, releasing, installing, configuring, updating, and adapting (as well as their counterparts of unreleasing, uninstalling, etc.). Most software deployment systems nowadays inherently support component-based software and handle dependencies among components. Popular examples of such systems are InstallShield [11], NetDeploy [15], and Tivoli [25].

For our purposes, however, most of these systems fall short in providing the necessary functionality. In particular, dependencies must be manually specified, they often assume all components are released by a single organization, and they provide limited support for managing multiple versions of a single component. SRM [28], the Software Dock [10], and RPM [1,19] are examples of systems that address these shortcomings. They explicitly support distributed and decentralized organization in releasing their components, and also include support for managing different versions of those components. Unfortunately, these software deployment systems operate in a vacuum: they are not connected to a configuration management system. This is detrimental in two ways: (1) they cannot obtain components from the configuration management system to release them and (2) they cannot "install" components into the configuration management system to bring them into the development environment. As a result, much manual effort is still required.

## 4 Approach

The key insight underlying our approach is depicted in Figure 2. Rather than a traditional model in which a component is first developed in a configuration management system and then deployed using a *separate* software deployment system, our approach is based upon a model in which a single configuration management system integrally supports both the development of a component and its deployment to other organizations. This allows both types of functionality to take advantage of each other and provides users with a single, integrated solution.

To support this integration, TWICS builds upon an existing configuration manage-



**Figure 2.** Traditional Separation of CM and Deployment Functionality versus TWICS.

ment system, Subversion [24], and overlays it with functionality for explicitly managing and deploying components. In particular, as described in detail in the next section, TWICS organizes the configuration management repository in a component-based fashion and adds three extra modules for managing the resulting artifacts.

- A *producer package manager*, which supports a developer in publishing a component;
- A *component publisher*, which supports other organizations in downloading a component; and
- A *component downloader*, which downloads a component and places it under local configuration management control.

All these modules take into account the possibility of dependencies. Additionally, they collaborate in establishing and enforcing different trust policies as they exist among different organizations.

The resulting architecture of TWICS is shown in Figure 3. The *policies* component plays a key role in this architecture. It contains the declarative data describing the trust policy of a particular organization. For instance, in case of the organization FIRSTFORMAT of Section 2, its *policies* component may state that the organization SPELLONE may download but not redistribute its components. Upon a download request, the *component publisher* consults the *policies* component to ensure that only trusted parties are able to download the components.

Of note is that it cannot be expected that all organizations use the same configuration management system (e.g., Subversion). TWICS is therefore built on top of a generic API (see Figure 4). Given a bridge from this API to a particular configuration management system, TWICS can operate on top of that configuration management system. We purposely kept the API small such that TWICS can easily be ported onto new CM systems.

## 5 Implementation

We have implemented a prototype version of TWICS according to the architecture described in Section 4. Below, we discuss each of the three major components in the architecture, namely the *producer package manager*, *component publisher*, and *component downloader*. Before we do so, however, we first discuss how TWICS structures the normally unstructured repository of Subversion such that it provides a component-based orientation.

### 5.1 Repository Structure

TWICS configures the repository structure of its underlying configuration management system in such a way as to enable a transparent development and deployment of

both internal and external components. The resulting repository structure is shown in Figure 5. TWICS partitions the available versioning space into two parts: a separate module called “TWICS”, in which it stores all the data pertaining to its own operation, and the remaining versioning space, in which actual development takes place.

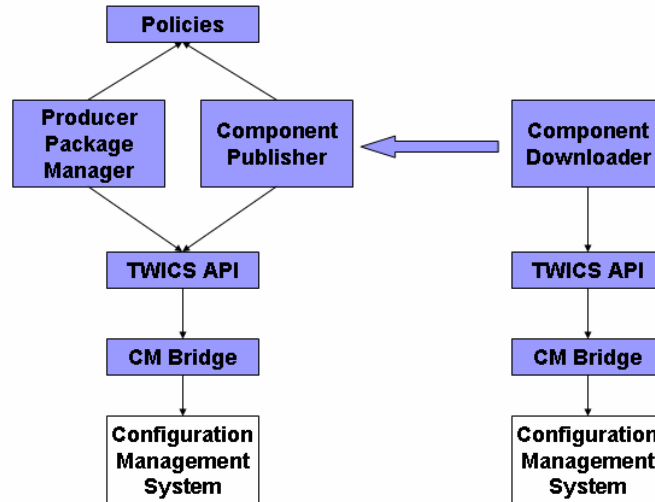


Figure 3. Architecture of TWICS.

```

Producer side API:
// Method to create source and binary packages and place them on the TWICS side
createComponent(String componentName, String componentVersion);
// Method to add dependency and metadata information for a deployable package
addMetaData(String componentName, String componentVersion);
// Method to get the package to be deployed from the TWICS repository
getPackage(String componentName, String componentVersion, String accessAllowed, String redistributionPermissions);
// Method to configure the CM repository to accommodate TWICS
configureSVN();

Consumer side API:
// Method to deploy an external component in the local TWICS repository
putComponent(String componentName, String componentVersion);
// Method to read the metadata of an external component and know its dependencies
getMetaData(String componentName, String componentVersion);
  
```

Figure 4. TWICS Internal API.

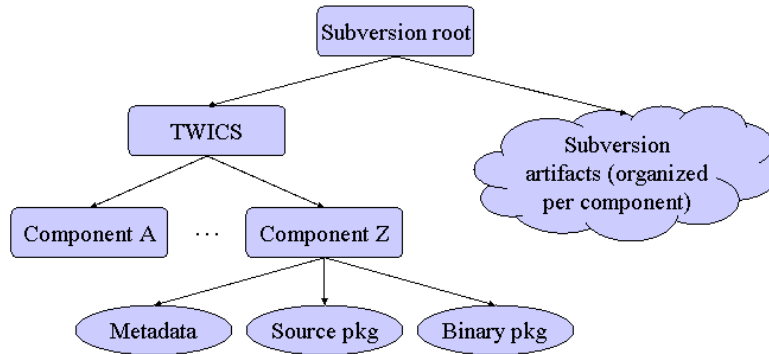


Figure 5. TWICS Repository Structure.

To preserve integrity, users should not directly manipulate the TWICS part of the repository, but use the TWICS user interface instead. The remaining versioning space, however, can be manipulated at will, provided that users adhere to a naming scheme in which each Subversion module represents a component.

The TWICS part of the repository serves as a controlled store for packaged components. These components may have been developed locally, in which case they are stored for download by remote organizations. They may also have been developed remotely, in which case the controlled store serves as a staging area from which components are unpacked and brought into the development side of the repository. Components are stored as archives, and may be available as a source archive, a binary archive, or both. The metadata of a component is stored alongside the component itself, containing both data describing the component (e.g., name, version, authoring organization, dependencies) and data concerning the trust policy to be applied (e.g., organizations permitted to download the component, redistribution permissions).

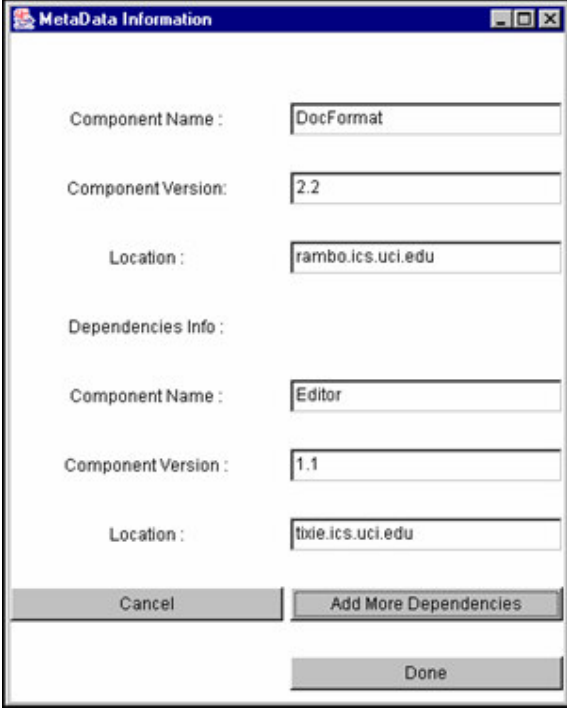
Using the native versioning mechanisms of Subversion, different versions of a component can be stored in the TWICS repository. This allows different versions of a locally developed component to be available for download, and furthermore supports an organization in obtaining and separating multiple versions of an externally developed component.

Separating the TWICS repository from the actual development area helps to separate the remote evolution of an external component from any local changes that may have been made. When a new version of such an external component is downloaded, placing it in the TWICS repository does not disturb any local development. At a later time, suitable to the local organization, can the remote changes be merged and integrated into the local effort.

Note that, under this scheme, different instances of TWICS in effect act as peers to each other. Each can simultaneously serve as an instance that makes components available to other instances and as an instance that downloads components from other instances.

## 5.2 Producer package manager

Once a component has been developed using normal configuration management procedures (e.g., using Subversion), it must be packaged and made available for release. To do so, a developer uses the TWICS *producer package manager* component. This component, shown in Figure 6, first requests some metadata from the developer describing the component. In particular, it requests the name and version of the component to be released, the server to which it should be released (defaulted to the current configuration management repository), and any dependencies that the component may have on other components. After this information has been filled out, TWICS automatically creates a source and a binary distribution for the component and places them on the TWICS side of the repository. Currently, TWICS assumes Java components and creates a package by automatically including source files in the source package and class files in the binary package. Future versions of TWICS will extend this capability significantly.



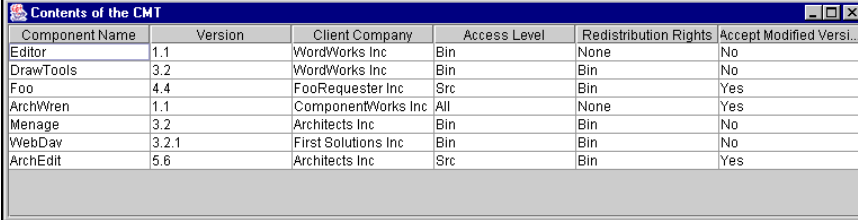
Field	Value
Component Name :	DocFormat
Component Version:	2.2
Location :	rambo.ics.uci.edu
Dependencies Info :	
Component Name :	Editor
Component Version :	1.1
Location :	tixie.ics.uci.edu

Buttons: Cancel, Add More Dependencies, Done

**Figure 6.** Releasing a Component.

The second step that a developer must take is to attach a deployment policy to the newly released component. As shown in Figure 7, a developer can specify (on a per-organization basis, if needed) the level of access and redistribution rights of outside organizations. This includes whether the component may be downloaded in source

and/or binary form as well as whether the component may be modified and/or redistributed directly by the outside organization. Note that if redistribution is disallowed, the outside organization can still use the component and upon release of its own components establish a remote dependency.



Component Name	Version	Client Company	Access Level	Redistribution Rights	Accept Modified Versi...
Editor	1.1	WordWorks Inc	Bin	None	No
DrawTools	3.2	WordWorks Inc	Bin	Bin	No
Foo	4.4	FooRequester Inc	Src	Bin	Yes
ArchWren	1.1	ComponentWorks Inc	All	None	Yes
Menage	3.2	Architects Inc	Bin	Bin	No
WebDav	3.2.1	First Solutions Inc	Bin	Bin	No
ArchEdit	5.6	Architects Inc	Src	Bin	Yes

Figure 7. Specifying Trust Policies.

### 5.3 Component Publisher

Once the trust policy has been specified, the component is available to other organizations for download. The download process itself is governed by the *component publisher* component, whose sole responsibility is to make sure the consumer is allowed the request they are making. If not, the component cannot be downloaded. If the request can be granted, the component, along with its metadata, is shipped to the consumer.

### 5.4 Component Downloader

The *component downloader* component eases the process of obtaining remotely developed components. Based on the input received from the user, the component downloader sends a deployment request to the producer of the requested component. If the user is indeed allowed to download the component, an archive will be received from the producer and placed on the TWICS side the configuration management repository. After that, the component downloader checks the metadata of the component and automatically fetches any required dependencies. This process is performed recursively, until all the dependencies are obtained and placed in the local environment. Note that if a component that is a dependency is already at the consumer site, it will not be downloaded again.

After the components have been downloaded, TWICS supports a user in automatically unpacking the archives and placing the components on the development side of the repository. From there on, a user can start using the components.

Note that if a user locally modified a component for which they are now downloading a new version, TWICS currently requires them to manually integrate the two (perhaps with assistance from the standard merge routines in Subversion). While TWICS makes

sure to not overwrite any changes, automated support for detecting and resolving these situations is certainly desired (see Section 5.5).

## 5.5 Discussion

TWICS is a work in progress. The prototype described above represents only the beginnings of our efforts. While it demonstrates the basic feasibility of the idea of merging configuration management and software deployment functionality into a single system, functionality wise the system is still in its infancy. At the forefront of our efforts at improving TWICS are the following:

*Incorporate extensive deployment support.* TWICS should leverage systems such as SRM [28], for automatically tracking and managing dependencies, and the Software Dock [10], for supporting the configuration of newly downloaded components. Currently, those tasks have to be performed by hand, which is cumbersome and time consuming.

*Support the specification and enforcement of advanced trust policies.* At the moment, our use of the word “trust” is somewhat overstated given the simplicity of the policies involved. In future, we plan to extend TWICS to incorporate support for specifying, in much more detail, the particular relationships between different organizations. Additionally, we will enhance the TWICS infrastructure to guarantee the desired properties. At present, for example, a consumer can circumvent a “do not redistribute” mandate by taking a component out of the configuration management repository and storing it under a different name. Such functionality should be disallowed.

*Enhance the packager and unpackager components.* Currently, these components have limited functionality and make a number of simplifying assumptions. We would like to enhance the packager by providing a user interface for creating rules according to which files should be chosen to be incorporated in a package. Similarly, we would like to enhance the unpackager by incorporating some tracking support that: (a) maintains the version number originally assigned by the component producer, (b) detects if the consumer has made any local changes, and (c) automatically integrates the new version with any local changes.

## 6 Conclusion

TWICS represents a first attempt at building a configuration management system that addresses the problem of managing the evolution of distributed, inter-related components. Set in the world of decentralized collaborating organizations, TWICS supports component-based software development via an integrated configuration management and software deployment system that establishes bi-directional communication among the organizations that produce components and the organizations that consume those components. A key aspect of TWICS is that it treats producers and consumers in the same way, and that its underlying repository structure is the same for both. In effect,

producers and consumers serve as peers to one another in a distributed, loosely coupled, federated configuration management repository.

Although TWICS is a limited prototype, it is worth noting that it provides an answer to each of the questions posed in Section 2. By incorporating deployment functionality directly in the configuration management system, components that are developed by external organizations can be brought into the local development environment. By providing a controlled store separate from the development versioning space, the remote evolution of components is separated from their local evolution. By supporting the specification of dependencies and always maintaining the metadata alongside a component, a component-based software system can be effectively deployed. And finally, by establishing trust policies among the different configuration management repositories participating in a federation, we can address the autonomy, privacy, and trust issues that are key to effectively provide configuration management for decentralized component-based software development. We intend to continue developing TWICS to further address these questions.

## Acknowledgments

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Effort also partially funded by the National Science Foundation under grant number CCR-0093489. 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 Laboratory, or the U.S. Government.

## References

- [1] E.C. Bailey, *Maximum RPM*. Red Hat Software Inc., 1997
- [2] B. Berliner. *CVS II: Parallelizing Software Development*. Proceedings of the USENIX Winter 1990 Technical Conference, 1990: p. 341-352
- [3] C. Burrows and I. Wesley, *Ovum Evaluates Configuration Management*. Ovum Ltd., Burlington, Massachusetts, 1998
- [4] ComponentSource, <http://www.componentsource.com/>, 2002
- [5] R. Conradi and B. Westfechtel, *Version Models for Software Configuration Management*. ACM Computing Surveys, 1998. 30(2): p. 232-282
- [6] M. de Jonge. *Source Tree Composition*. Proceedings of the Seventh International Conference on Software Reuse, 2002
- [7] L.G. DeMichiel, L.U. Yalcinalp, and S. Krishnan, *Enterprise Java Beans Specification, Version 2.0*, <http://java.sun.com/products/ejb/docs.html>, 2001

- [8] J. Estublier and R. Casalles, *The Adele Configuration Manager*, in Configuration Management, W.F. Tichy, Editor. 1994: p. 99-134
- [9] P.H. Feiler. *Configuration Management Models in Commercial Environments*. Software Engineering Institute, Carnegie Mellon University, 1991
- [10] R.S. Hall, D.M. Heimbigner, and A.L. Wolf. *A Cooperative Approach to Support Software Deployment Using the Software Dock*. Proceedings of the 1999 International Conference on Software Engineering, 1999: p. 174-183
- [11] InstallShield, <http://www.installshield.com/>, 2001
- [12] M. Larsson and I. Crnkovic. *Configuration Management for Component-based Systems*. Proceedings of the Tenth International Workshop on Software Configuration Management, 2001
- [13] M.D. McIlroy, *Mass Produced Software Components*. Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee, P. Naur and B. Randell (eds.), 1968: p. 138-155
- [14] T. Mens, *A State-of-the-Art Survey on Software Merging*. IEEE Transactions on Software Engineering, 2002. 28(5): p. 449-462
- [15] NetDeploy, <http://www.netdeploy.com/>, 2001
- [16] J. Oberleitner. *The Component Workbench: A Flexible Component Composition Environment*. M.S. Thesis, Technische Universität Vienna, 2001
- [17] Object Management Group, ed. *The Common Object Request Broker: Architecture and Specification*. 2001, Object Management Group
- [18] D.S. Platt, *Introducing Microsoft Dot-Net*. Microsoft, Redmond, 2001
- [19] Red Hat, <http://www.rpm.org/>, 2001
- [20] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998
- [21] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998
- [22] Telelogic, *CM/Synergy*, <http://www.telelogic.com/products/synergy/cmsynergy/index.cfm>, 2002
- [23] W.F. Tichy, *RCS, A System for Version Control*. Software - Practice and Experience, 1985. 15(7): p. 637-654
- [24] Tigris.org, *Subversion*, <http://subversion.tigris.org/>, 2002
- [25] Tivoli Systems, <http://www.tivoli.com/>, 2001
- [26] V. Traas and J. van Hilleegersberg, *The Software Component Market on the Internet: Current Status and Conditions for Growth*. Software Engineering Notes, 2000. 25(1): p. 114-117
- [27] E. Tryggeseth, B. Gulla, and R. Conradi. *Modelling Systems with Variability Using the PROTEUS Configuration Language*. Proceedings of the International Workshop on Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, 1995: p. 216-240
- [28] A. van der Hoek, 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, 1997: p. 159-175

- [29] D. Wiborg Weber. *Requirements for an SCM Architecture to Enable Component-based Development*. Proceedings of the Tenth International Workshop on Software Configuration Management, 2001