A Cooperative Approach to Support Software Deployment Using the Software Dock

Technical Report CU-CS-871-98

Richard S. Hall, Dennis Heimbigner, Alexander L. Wolf
Department of Computer Science
University of Colorado
Campus Box 430
Boulder, CO 80309 USA
303-492-8115
[rickhall,dennis,alw]@cs.colorado.edu

ABSTRACT
Software deployment is an evolving collection of interrelated processes such as release, install, adapt, reconfigure, update, activate, deactivate, remove, and retire. The connectivity of large networks, such as the Internet, is affecting how software deployment is being performed. To take full advantage of this connectivity, new software deployment technologies must be introduced in order to support these processes. The Software Dock research project is creating a distributed, agent-based deployment framework to support the ongoing cooperation and negotiation among software producers themselves and among software producers and software consumers. This deployment framework is enabled by the use of a standardized semantic schema for describing software systems, called the Deployable Software Description (DSD) format. The Software Dock employs agents to traverse between software producers and consumers and to perform software deployment activities by interpreting the semantic descriptions of the software systems. The Software Dock infrastructure enables software producers to offer high-level deployment services that were previously not possible to their customers.

Keywords
Software deployment, Java, mobile agents, configuration management

1 INTRODUCTION

The connectivity of large networks, such as the Internet, is affecting how software deployment is being performed. The simple notion of providing a complete installation procedure for a software system on a CD-ROM is giving way to a more sophisticated notion of ongoing cooperation and negotiation among software producers and consumers. This connectivity and cooperation allows software producers to offer high-level deployment services that were previously not available to their customers. In the past, only software system installation was widely supported, but already support for the update process is becoming common. Support for other software deployment processes, though, such as release, adapt, activate, deactivate, remove, and retire [see Section 2] is still virtually non-existent.

As new enabling technologies become available, software producers are slowly accepting more of the shared responsibility for the long-term operation of their software systems. In order to fully support software deployment, these enabling technologies must:

- operate on a variety of platforms and network environments, ranging from single sites to the entire Internet,
- provide a semantic model for describing a wide range of software systems in order to facilitate some level of software deployment process automation,
- provide a semantic model of target sites for deployment in order to describe the context in which deployment processes occur, and
- provide decentralized control for both software producers and consumers.

The Software Dock research project is addressing these concerns. The Software Dock is a system of loosely coupled, cooperating, distributed components. The Software Dock supports software producers by providing the release dock that acts as a repository of software system releases. At the heart of the release dock is a standard semantic schema for describing software systems. The field dock supports the consumer by providing an interface to the consumer's resources, configuration, and deployed software systems. The Software Dock employs agents that travel from release docks to field docks in order to perform specific software deployment tasks while docked at a field dock. The agents perform their tasks by interpreting the semantic descriptions of both the software systems and the target consumer site description. A wide-area event system

connects release docks to field docks and enables asynchronous, bi-directional connectivity.

The purpose of this paper is to discuss how the Software Dock project supports software deployment processes. This is accomplish by first introducing the processes that comprise software deployment. Section 3 provides a high-level introduction of the Software Dock architecture, while Section 4 describes the Deployable Software Description (DSD) format, a critical piece of the Software Dock project used to semantically describe software systems. Section 5 discusses specific deployment process support through the use of agents. Section 6 discusses security and electronic commerce as it relates to the deployment and the Software Dock specifically, while Section 7 discusses related work. Lastly, the current status and future work is discussed in Sections 8 and 9, respectively, followed by the conclusion.

2 SOFTWARE DEPLOYMENT LIFE CYCLE PROCESSES

In the past, software deployment has largely been defined as the installation of a developed software system; this view of software deployment is incomplete. Software deployment is actually a collection of interrelated activities that form the *software deployment life cycle*. The software deployment life cycle, as we have defined it, is an evolving definition that consists of the following processes: release, retire, install, activate, deactivate, reconfigure, update, adapt, and remove. Each process pertains to either a producer-side or a consumer-side activity and is described in more detail below.

Producer-side Processes

One of the main deployment concerns of the software producer is that of the ***release*** process. The release process is the bridge between development and deployment. It encompasses all the activities needed to package, prepare, provide, and advertise a system for deployment to consumer sites. The release package that is created not only consists of the physical artifacts that comprise a given software system, it must also consist of a semantic description of the software system in order to enable automated processing. As modification or updates are made to the software system, the software producer must repeat the release process to create an updated release package.

When a software producer is no longer able or willing to support a given software system, it is necessary to perform the ***retire*** process. The retire process withdraws support for a software system or a given configuration of a software system. The retire process should not be confused with the consumer-side remove process; retiring a software system makes it unavailable for future deployment, but it does not necessarily affect consumer sites where the retired software system is currently deployed. Consumers of the software may continue to use the software without knowing that it has been retired, but the retire process should attempt to notify current users that support for the software system is being withdrawn.

Consumer-side Processes

The ***install*** process is the initial deployment activity performed by a consumer. The install process must configure and assemble all of the resources necessary to use a given software system. The install process uses the package created in the release process above. For a specific package, the install process interprets the encoded knowledge and then examines the target consumer site in order to determine how to properly configure the software system for the specific site. Once installation is completed the deployed software system is ready for use and is ready for other deployment activities.

After a software system is installed, the ***activate*** and ***deactivate*** processes allow the consumer to actually use the software system. The activate process is responsible for running or executing a deployed software system. For a simple tool, activation involves establishing some form of command (or clickable graphical icon) for executing the binary component of the tool. For a distributed system, there may be multiple components that need to be running in order for the system to be usable. The deactivate process is the inverse of the activate process. It is responsible for shutting down any executing components of an activated software system.

Throughout the lifetime that a software system is installed at a consumer site, it is not a static entity with respect to software deployment. Instead, the ***reconfigure***, ***update***, and ***adapt*** processes are responsible for changing and maintaining the deployed software system configuration. These processes may occur in any order and any number of times.

The update process modifies a previously installed software system. The main purpose of update is to deploy a new, previously unavailable configuration of a software system. An update becomes necessary when a software producer makes changes to the semantic description of a deployed software system. The changes to the semantic description may denote a new version of the software system, a content update, or simply a description update.

The reconfigure process also modifies a previously installed software system, but its purpose is to select a different configuration of a deployed software system from its existing semantic description.

The purpose of the adapt process is to maintain the consistency of the currently selected configuration of a deployed software system. The adapt process must monitor changes at the consumer site and respond to those changes in order to maintain consistency in the deployment software system. Adaptation becomes necessary when a change is made to the local consumer site that affects the deployed software system.

Once a software system is no longer required at a consumer site, the ***remove*** process is performed. The remove process must undo all of the changes to the consumer site that may have been caused by previous deployment activities for a given software system. Special attention has to be paid to shared resources such as data files and libraries in order to prevent dangling references to a required resource. As a result, the remove process must examine the current state of the consumer site, its dependencies, and constraints, and then remove the software package in such a way as to not violate these dependencies and constraints.

3 SOFTWARE DOCK ARCHITECTURE

The Software Dock research project, originally described in [8], addresses support for software deployment processes by creating a framework that enables cooperation among software producers themselves and between software producers and software consumers. In order to provide such a framework, the Software Dock architecture [see Figure 1] defines components that represent these two participants in the software deployment problem space. The *release dock* represents the software producer and the *field dock* represents the software consumer. In addition to these components the Software Dock employs *agents* to perform specific deployment process functionality and a *wide-area*

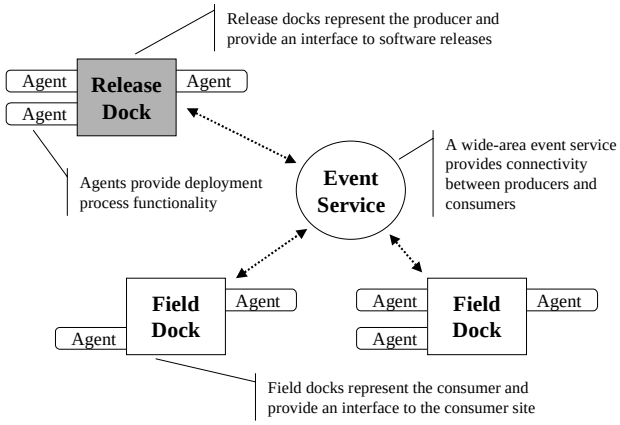

Figure 1: Software Dock Architecture

event system to provide connectivity between the release docks and the field docks.

In the Software Dock framework, the release dock is a server that resides within a software producing organization. The purpose of the release dock is to serve as a release repository for the software systems that the software producer provides. The release dock provides a Web-based release mechanism that is not wholly unlike the release mechanisms that are currently in use; it provides a browser-accessible means for software consumers to browse and select software for deployment.

The release dock, though, is more sophisticated than most current release mechanisms. Within the release dock, each software release is semantically described using a standard semantic schema; this standard semantic schema is presented in more detail in the next section. Each software release is accompanied with generic agents that perform software deployment processes by interpreting the semantic description of the software release. The release dock provides a programmatic interface for agents to access its services and content. Finally, the release dock generates events as changes are made to the software releases that it houses. Agents associated with deployed software systems can subscribe for these events in order to be informed when specific deployment processes may be necessary, such as an update.

The field dock is a server that resides at a software consumer site. The purpose of the field dock is to serve as an interface to the consumer site. This interface provides information about the state of the consumer site's resources and configuration; this information provides the context into which software systems from a release dock will be deployed. Agents that accompany software releases "dock" themselves at the target consumer site's field dock. The interface provided by the field dock is the only interface that an agent has to the underlying consumer site. This interface includes capabilities to query and examine the resources and configuration of the consumer site; examples of each might include installed software systems and the operating system configuration.

The release dock and the field dock are very similar components. Both are servers where agents can "dock" and perform activities, and both house a standardized, hierarchical registry of information that records the configuration or the contents of their respective sites and create a common namespace within the framework. The registry model used in both the release and field docks is nested collections of attribute-value pairs, where the nested collections form a hierarchy. Any change to the registry generates an event that agents may receive in order to perform subsequent activities. The registry of the release dock mostly provides a list of available software releases, whereas the registry of the field dock performs a much more valuable role.

Consumer-side information is critical in performing nearly any software deployment process. In the past software deployment was complicated by the fact that consumer-side information was not available in any standardized fashion. The field dock registry addresses this issue by creating a detailed, standardized, hierarchical schema for describing the state of a consumer site. By standardizing the information available at a consumer site, the field dock creates a common software deployment namespace for accessing consumer site properties, such as operating system and computing platform information. This information, when combined with the semantic description of a software system, can be used to perform specific software deployment processes. As such, the two semantic descriptions can be

considered two halves of a whole.

Agents are used to perform the actual software deployment process functionality. When a software system is requested to be installed on a given consumer site, initially only an agent responsible for installing the specific software system and the semantic description of the specific software system are loaded onto the consumer site from the originating release dock. The installation agent docks at the local field dock and uses the semantic description of the software system and the consumer site state information provided by the field dock to configure the selected software system. When the agent is done configuring the software system for the specific target consumer site, it requests the precise configuration that it requires from its release dock.

As part of the initial installation process, the installation agent may request other agents from its release dock to come and dock at the local field dock. These other agents are responsible for other deployment activities, such as update, adapt, reconfigure, and remove. Each agent performs its associated process by interpreting the semantic information of the software system description and the consumer site configuration.

The wide-area event service in the Software Dock architecture provides a means of connectivity between software producers and consumers for "push"-style capabilities. Agents that are docked at remote field docks can subscribe for events from other release docks and can then perform subsequent actions in response to those events, such as performing an update. Direct communication between agents and release docks is provided by standard protocols over the Internet. Both forms of connectivity combine to provide the software producer and consumer the opportunity to cooperate in their pursuit of software deployment process support.

4 DEPLOYABLE SOFTWARE DESCRIPTION (DSD) FORMAT

In order to automate or simplify software deployment processes it is necessary to have some form of semantic knowledge about the software systems being deployed. One approach to this requirement is the use of a standardized language or schema for describing a software system; this is the approach adopted by the Software Dock research project. In such a language or schema approach it is common to model software systems as collections of properties, where semantic information is mapped into standardized properties and values.

Minimally five classes of semantic information have been identified [6] that must be described by the software system model. These classes of semantic information are:

- **Assertions** – describe constraints on consumer-side properties that must be true otherwise the specific deployment process fails, such as supported hardware platforms or operating systems.
- **Dependencies** – describe constraints on consumer-side properties where a resolution is possible if the constraint is not true, such as installing dependent subsystems or reconfiguring operating system parameters.
- **Configuration** – describes relationships inherent in the software system itself, such as revisions and variants, and describes the deployment interfaces provided by the software system.
- **Artifacts** – describe the actual physical artifacts that comprise the software system.
- **Activities** – describe any specialized activities that are outside of the purview of standard software deployment processes.

The Software Dock project has defined the Deployable Software Description (DSD) format to address these semantic description needs. DSD is a critical piece of the Software Dock research project that is used to create generic software deployment process definitions.

DSD provides a standard semantic schema for describing a software system family. In this usage, a family is defined as all revisions and variants of a specific software system. The software system family was chosen as the unit of description, rather than a single revision, variant, or some combination, because it provides flexibility when specifying dependencies, enables description reuse, and provides characteristics, such as extending revision lifetime, that are necessary in component-based development.

The family description in DSD is broken up into multiple elements that address the five semantic classes of information described above. The sections of a DSD family description are identification, imported properties, system properties, property composition, assertions, dependencies, artifacts, interfaces, notifications, and activities. Some of these sections map directly onto the five semantic classes of information, others, such as system properties, property composition, interfaces, and notifications, combine to map onto the configuration class of semantic information. The identification section is largely human-readable content and is not for processing.

A DSD family description is a simple, hierarchical schema that is built around the notion of properties of the software system being described. For example, a typical property of a software system might be a version number. By defining such a property in a family description it is possible to organize the other pieces of the family description, such as assertions, dependencies, and artifacts, with respect to a given version number. Other examples of software system properties are performance variants and optional capabilities. Once the properties of a software system are defined then the property composition section describes the relationships between properties. For example, one property may include or exclude another property or may require secondary property selections. The composition rules de-

scribe valid configuration for the software system being described.

The remaining DSD family description sections are guarded by arbitrary boolean property expressions that indicate whether a specific schema element is applicable to a specific configuration. The property expression guards can be expressions over software system properties, consumer site properties, or both.

The following examples depict portions of a DSD description that describes a software system that has optional online help documentation. To describe the optional online help documentation, a software system property to represent the inclusion of the documentation is created:

```
Property {
  Name = "Online Help"
  Type = "Boolean"
  Description = "Include online help."
  … }
```

The above property definition creates a boolean property of the software system that is to be used for determining whether the online help documentation is applicable to a given configuration of the software system.

Also consider that the software system being described only supports the Solaris and Window 95 operating systems. To guarantee that these constraints are true an assertion is created:

```
Assertion {
  Condition = "($OS$ == 'Solaris') ||
              ($OS$ == 'Win95')"
  Description = "Test for supported
                 operating system."
  … }
```

This assertion tests the target consumer site's operating system properties by using the standard namespace that is created by the field dock registry. In the above assertion example, the variable `$OS$` is actually shorthand introduced for brevity; the actual variable is the standard field dock registry path expression of :

`$/Local/Software/OperatingSystem/Name$`.

The artifacts that comprise the online help documentation must also be described:

```
Artifacts {
  Guard = "($Online Help$ == true)"
  Artifact {
    Guard = "($OS$ == 'Solaris')"
    Signature = "a4ca443b8902d3410ec832"
    Type = "DOCUMENTATION"
    SourceName = "help.html"
    Source = "/proj/doc"
    DestinationName = "help.html"
    Destination = "doc"
    Mutable = false
    … }
  Artifact {
    Guard = "($OS$ == 'Win95')"
    Signature = "9283cd2378102f1a3b12ee"
    Type = "DOCUMENTATION"
    SourceName = "help.hlp"
    Source = "/proj/doc"
    DestinationName = "help.hlp"
    Destination = "doc"
    Mutable = false
    … } }
```

The artifacts are described by nesting them in an artifact collection. The above artifact collection is guarded by a property expression that tests the applicability of the artifact collection with respect to a specific configuration; in this case, the artifact collection is only applicable if the "Online Help" property of the software system is true. The actual online help documentation artifacts are described within the artifact collection, each of which are guarded by property expressions that test for a specific consumer site operating system value. The end result is that the proper artifact is installed with respect to the target consumer site and the selected configuration of the software system.

As a note, software system properties are arbitrary names; they have no meaning within DSD. Therefore, a property such as "version" has no special significance in DSD as it might in other configuration management disciplines. One result of this approach is that properties can be used to organize a software system in a variety of ways. For example, properties can be mapped to the traditional configuration management view of versions, the components in the software system architecture, or the features or capabilities of the software system.

5 SOFTWARE DOCK AGENTS

Agents play a pivotal role in the Software Dock project. Most of the other components in the Software Dock architecture are relatively passive elements, such as data and interfaces. Agents, on the other hand, are responsible for performing the functionality of nearly all of the software deployment processes.

One goal of the Software Dock is to provide a collection of generic agents that perform many of the standard software deployment processes, such as install, update, adapt, reconfigure, and remove. The agents, though useful in many cases, may not be sufficient for every case and therefore may also be used as base classes for the creation of other, more specialized deployment agents.

Agents perform their deployment processes by encoding some functionality that is then parameterized by the semantic information provided in the software family descriptions and the consumer site descriptions. In this fashion a single agent definition can be used for any software system de-

scribed using DSD and that software system can be deployed to any consumer site that has a field dock.

The Generic Agent Process

As described in Section 4, DSD models a software system based on properties and the proper configuration of those properties. A result of this approach led to the discovery of an abstract deployment process definition.

Many of the software deployment processes can be described as a modification to the values of the properties of a given software system. This valid set of software system properties defines a particular configuration of the software system. Once a configuration is determined it is possible to determine the applicable elements of the software family description. At this point a software deployment process only needs to make the deployed software system correspond to the applicable schema description elements. For example, if the version of a software system is changed from "1.0" to "1.1," then all of the artifacts associated with version "1.0" must be removed and the artifacts associated with version "1.1" must be added.

In general, the install, update, reconfigure, adapt, and remove software deployment processes all follow this same abstract algorithm.

Specific Agent Processes

The install agent is different from many of the other software deployment process agents because it is not working with an existing software system configuration. The install agent must determine the configuration of the software system to be installed by querying the field dock for necessary consumer site properties. In order to determine the values for software system properties, such as version or optional capabilities, the install agent may ask the consumer [see Figure 2]. Once a configuration is determined the install agent only needs to perform the actions associated with all of the applicable schema elements for the selected configuration, such as testing assertions, resolving dependencies, and retrieving artifacts.

Figure 2: Configuration Editor

The update agent deploys a new, previously unavailable configuration of a deployed software system. The new configuration is provided in a new semantic description of the software system that the update agent retrieves from its release dock. The update process must account for a previously deployed configuration of a software system. The update process may either be specifically directed by the "push" of a new configuration, such as a new version, or it may be undirected in the case of a "pull" update where a new configuration must be discovered or specifically selected by the user. An update, though, is not always the result of a change to the currently selected configuration, an update may only be a content update. In such a scenario, the update does not change the selected configuration of the software system, rather the content of the current configuration is updated. This is typical in many software systems that use a "channel" or content delivery model. In either case the update agent performs differential processing of the applicable schema elements, undoing the schema elements corresponding to the prior configuration if necessary and performing the activities associated with the schema elements of the new configuration. Any schema elements that are shared among configurations are left untouched.

The reconfigure agent allows the current configuration of a deployed software system to be changed. The changes that are allowed, though, do not include any new changes that have been made to the software system description on the release side; these types of changes are considered to be an update. As such, the reconfigure agent manipulates the existing semantic family description of a previously deployed software system. Once a new configuration is chosen from the existing family description, the reconfigure agent performs differential processing on the applicable schema elements much like the update agent.

The adapt agent tries to maintain the consistency of the currently selected configuration of a software system in the context of the consumer site. The adapt agent monitors events that might affect the deployed software system and takes an appropriate action when such events occur. The adapt agent may operate in a "pull" mode as well. In this mode, the adapt agent re-verifies the deployed configuration; for example, it rechecks assertions and dependencies, and it validates all of the artifacts. In either mode of operation the adapt agent attempts to resolve any problems it encounters.

The remove agent is responsible for removing a deployed configuration from a consumer site. The remove agent must ensure that no constraints are violated by the removal of the software system. For example, if other deployed software systems depend on the software system that is being removed, the remove should fail. The remove agent is also responsible for removing any dependent subsystems that it may have installed during its own deployment, if necessary.

There is an interesting, implicit issue with respect to all of the agents described above. All of the agents manipulate the schema description of a given software system in isolation of the software system itself. This means that an agent only needs the description of a software system to perform

most of its tasks. As a result, an agent can be much more efficient, especially in the area of transfer time, since by manipulating the schema description first, the agents only need to request exactly what they need to finish their tasks. This is possible since the release dock works in cooperation with the agents to perform the deployment processes.

6 SECURITY AND ELECTRONIC COMMERCE

Security and electronic commerce have an impact on the Software Dock research, but they have not been primary research issues. Despite this fact, these issues have not been summarily excluded in the solution discussed thus far.

Mobile agents cause a large security concern because they come from unknown sources. In order to address some of the security concerns in the Software Dock, agents operate in the Java Virtual Machine (JVM) sandbox. The field dock is the only local interface that an agent has to perform its tasks. To extend the interface provided to agents, the field dock uses a capability approach. The capability approach provided by the field dock allows certain restricted operations, such as controlled access to the disk. Currently, the JVM does not support a true capability approach, but this functionality is expected in the next release of Java. Regardless, all current agents are implemented as though this approach was in effect; thus there is a relatively simple transition when support for the capability-based security approach is released. In addition, this approach can be extended to adopt a mechanism by which agents can become trusted entities. In such a scenario, trusted agents may be provided with even more sensitive capabilities.

The Software Dock framework is also open to electronic commerce considerations. The Software Dock can easily be extended by the creation of additional agents at the release and field sites. As such, agents could be created to keep track of licensing issues. From a release perspective, agents could monitor each time a software system is installed or updated and then perform some procedure to charge a licensing fee to the consumer. Any variety of approaches is possible in this area, but none have been investigated since some scoping of the research area was necessary.

7 RELATED WORK

Since the scope of the Software Dock project is so large, there are many related technologies. This section only covers some of the most important related work. For some more detailed information on related technologies refer to [3] and [7].

The DSD schema created for the Software Dock project is not a unique attempt to create a standard schema for describing software systems. A handful of related technologies are also trying to address the same issue with similar approaches. Traditional configuration management modeling approaches, such as Adele [5] and PCL [21], have influenced DSD, particularly in the area of configuration selection. These traditional approaches, though, are more general configuration modeling languages that do not address software deployment in any fashion. In general, these approaches did not attempt to create a standard schema for any specific task, rather the modeling language was the primary contribution.

A more recent, high-profile effort to create a standard software deployment schema is called the Open Software Description (OSD) [9] format. This effort is an initial collaboration between Microsoft and Marimba to create a schema for describing software systems for "push" technologies. OSD is very premature and merely allows for the description of multiple coarse-grain variants of a single revision of a software system; dependent software systems may also be specified. The descriptive information includes some identification information and pointers to archives where the physical artifacts can be found. The resulting description is too simplistic to perform any significant software deployment automation.

The Desktop Management Task Force (DMTF) has created the Management Information Format (MIF) [4] for describing software systems. DMTF formed working groups to create standard syntax elements in MIF for describing various computing resources, including software systems. An extension to MIF has been created by Tivoli and is called the Application Management Specification (AMS) [19]. Since AMS is a superset of MIF, only AMS is discussed here. AMS is much more mature than OSD. AMS describes a single revision of a single variant of a software system in great detail. Software system composition, constraints, dependencies, identification, support, and artifacts are some of the elements that AMS describes. AMS is not intended, though, to automate all of the software deployment processes. Instead, AMS describes a semi-static configuration of a software system that is to be installed and monitored at a consumer site; the notion of manipulating internal software system properties like revisions or variants is not directly supported. It is also assumed that there is no cooperation between software producers and software consumers, rather there is a more centralized "administration" authority that is responsible for maintaining the state of deployed software systems.

The Defense Information Infrastructure Common Operating Environment (DII COE) [12] is a Department of Defense effort to restrict the set of components used to build their software systems. The COE supports, among other things, a standard means for packaging components for delivery and installation. These packages are called *segments* [13], where each segment is a separate, installable entity. The DII COE segment describes the constraints, dependencies, and artifacts of a software system. High-level software deployment process support is provided in the form of scripts, though all deployment activities are not directly supported. Like other approaches, the deployed

software system configurations are largely considered static entities that do not change or cannot be manipulated. The support provided is intended more for a centralized administration authority and there is no release-side support.

Other approaches, such as GNU Autoconf [15], try to resolve consumer site description by using scripts and heuristics to directly examine the state of a site, but these methods are not always accurate and they do not scale well. The Microsoft Registry [10], is a hierarchical registry of consumer site information for the Windows platform. The schema used in this registry is only partially standardized and even the standardized portions are not sufficient to semantically describe software systems for deployment.

The Redhat Package Manager (RPM) [1] is a tool for the Linux user community that provides many software deployment features. RPM packages contain the software system to be deployed and a semantic description of the software system; this description includes constraints, dependencies, artifacts, and activities in the form of scripts. The granularity of an RPM package is a single revision and a single variant. As a result, only limited forms of configuration selection are supported. RPM does not have a notion of a "release-side" and therefore is only able to request and manipulate complete packages. Also, RPM is intended more for single-site deployment and provides no support for multi-site deployment or management.

A host of install utilities exist in the commercial world, such as InstallShield [11]. These systems typically work real well for installation, but only address a handful of deployment processes, such as reconfigure and remove, in a limited form. Recent install utilities are starting to address the connectivity of the Internet, such as netDeploy [18] and PC-Install with Internet Extensions [22]. Some of these utilities are addressing the update process as well. In general, most of these solutions do not provide reasonable software system description capabilities. The level of semantic information is less declarative than necessary for generic software deployment automation.

Another class of commercial and research utilities exist to support artifact update; some of these systems include Castanet [16], NSBD [14], and rsync [20]. In most of these systems, there is little if any support for other software deployment processes. These solutions provide only a very simple model for describing software systems, in most cases a software system is merely considered to be a collection of files.

8 CURRENT STATUS

A prototype of the Software Dock deployment framework has been created. The Software Dock prototype has been implemented entirely in Java and uses Voyager [17] from ObjectSpace as an inter-process communication mechanism and a mobile agent enabling technology. A related research project at the University of Colorado, called SIENA [2], provides a wide-area event service.

An evolving definition of the DSD was created. The current definition of the DSD contains most of the main elements to support gross software deployment behavior.

The current implementation of the Software Dock infrastructure includes elements for both the release-side and the consumer-side. A release dock implementation has been created to house the various software system releases that a software producer has available. The creation of release packages for the release dock is supported by a schema editing tool. This simple schema editor provides a way to create and edit DSD descriptions of software systems and automates some tasks, such as the entry of software artifacts into the DSD description. Once a DSD description has been created with the schema editor, the new or updated release can be submitted to the local release dock and made available for deployment. The submission of a release to the release dock automatically generates a set of HTML pages for the new release that can be browsed by consumers and used to instigate the install process.

On the consumer-side a field dock has been created. The field dock describes various aspects of the consumer site, such as platform, operating system, memory, and resources. The field dock also provides a place for agents to "dock" and perform software deployment related tasks by providing an interface to the underlying consumer site. To further support the consumer-side, a tool, called a docking station [see Figure 3], has been created that provides an interface to the software systems that have been deployed at the consumer site. The docking station provides an interface to the

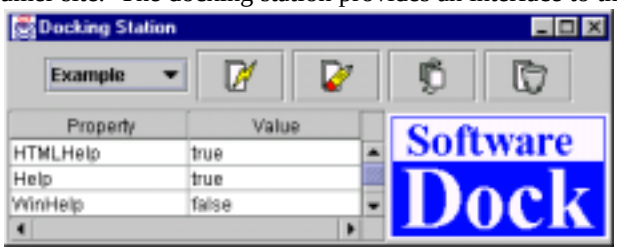

Figure 3: Docking Station Support Tool

deployment processes that can be performed on the locally deployed software systems. The docking station can be used to request updates, reconfigures, adapts, and removes.

A collection of generic agents has been created to interpret the DSD software system descriptions in order to perform specific software deployment processes. These generic agents include install, reconfigure, update, adapt, and remove. Each of these agents is fully parameterized by the DSD software description. All generically perform the configuration and selection process and then check assertions, resolve subsystem dependencies, and request and retrieve physical artifacts. The end result is support for the release and deployment of configurable content software systems.

The current implementation has been used in a demonstration to describe a Web content-based software system called the Online Learning Academy (OLLA) created by a division of Lockheed Martin. OLLA consists of 45 megabytes in over 1700 files. OLLA also demonstrated dependencies on two subsystems called Disco and Harvest. The software deployment processes of release, install, reconfigure, update, adapt, and remove have all been initially demonstrated using the generic agents described in this paper along with the DSD description of all three software systems.

9 FUTURE WORK

The current implementation of the Software Dock concentrates on the one-to-one aspects of the software producer/consumer relationship. There is no inherent limitation in the Software Dock framework for supporting other aspects of the software producer/consumer relationship. The most obvious scenario is that of the administrator role at a consumer site.

In order to support an administrator role, a new collection of "remote" agents will be created. These remote agents will behave much like the current agents, except that they will also be parameterized by consumer site names. With such a capability, an administrator will be able to specify that an activity, such as install or update, should occur on a specific site or a specific set of sites.

To further support the administrator role, a new server, called the interdock, will be introduced. An interdock server will contain more global information about the consumer organization, such as site domains and global services. With the interdock, some administration tasks will be simplified and it will also be possible to start to address more complicated deployment scenarios, such as those of distributed, coordinated software systems.

In addition, the DSD will continue to be extended and expanded. Support for administration policies will be enhanced. Arbitrary dependency specification, rather than just subsystem dependencies, will also be researched. Lastly, better support for specialized deployment activities will be investigated further.

10 CONCLUSIONS

Software deployment is not a single process, such as install, rather it is a collection of interrelated processes that must be performed after a software system has been developed and made available to consumers. Support for software deployment by software producers has been neglected until recently. Large network environments, such as the Internet, offer connectivity that can be used as an enabling technology for software producers to offer high-level software deployment services to their customers, services that were previously not possible. Combining the connectivity provided by large networks with other pieces of software deployment technology, a cooperative framework for supporting software deployment can be created. The Software Dock is creating such a framework.

The Software Dock supports software deployment processes by introducing components that represent software producers and consumers, release docks and field docks, respectively. The definition and use of a standard semantic schema for describing software systems is central to the Software Dock framework, and it provides, in a declarative form, all of the knowledge necessary to perform software deployment processes. Finally, agents are employed to embody the actual functionality of the deployment processes. The agents realize the deployment process functionality in a generic fashion by interpreting the declarative schema description of the software system.

ACKNOWLEDGMENTS

This material is based upon work sponsored by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

REFERENCES

1. E. C. Bailey. "Maximum RPM," Red Hat Software, Inc., ISBN: 1-888172-78-9, Feb. 1997.
2. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. "Design of a Scalable Event Notification Service: Interface and Architecture," Technical Report, Dept. of Computer Science, University of Colorado, 1998.
3. A. Carzaniga, A. Fuggetta, R.S. Hall, A. van der Hoek, D. Heimbigner, A.L. Wolf. "A Characterization Framework for Software Deployment Technologies," Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.
4. Desktop Management Task Force, "Software Standard Groups Definition, Version 2.0," Mar. 27, 1996. http://www.dmtf.org/tech/apps.html.
5. J. Estublier and R. Casallas. "The Adele Configuration Manager," Configuration Management, Wiley, 1994, pp. 99-134.
6. R. S. Hall, D. Heimbigner, and A. L. Wolf. "Requirements for Software Deployment Languages and Schema," Proceedings of the 1998 International Workshop on Software Configuration Management, July 1998.
7. R. S. Hall, D. Heimbigner, and A. L. Wolf. "Evaluating Software Deployment Languages and Schema," Proceedings of the 1998 International Conference on Software Maintenance, IEEE Computing Society, Nov. 1998.
8. R. S. Hall, D. Heimbigner, A. van der Hoek, A. L.

Wolf. "An architecture for Post-Development Configuration Management in a Wide-Area Network," Proceedings of the 1997 International Conference on Distributed Configurable Systems, IEEE Computing Society, May 1997, pp. 269-278.

9. A. van Hoff, H. Partovi, T. Thai. "The Open Software Description Format (OSD)," Microsoft Corp. and Marimba, Inc., 1997. http://www.w3.org/TR/NOTE-OSD.html.

10. Jerry Honeycutt. "Using the Windows 95 Registry," Que Publishing, Indianapolis, IN, 1996.

11. InstallShield Corp. InstallShield, 1998. http://www.installshield.com.

12. Joint Interoperability and Engineering Organization. "Defense Information Infrastructure Common Operating Environment Baseline Specificiations," Version 3.0, Defense Information Systems Agency, CM-400-25-05, Oct. 31 1996. http://spider.osfl.disa.mil/cm/baseline/base_line3/basel in3.pdf

13. Joint Interoperability and Engineering Organization. "How to Segment Guide," Version 4.0, Defense Information Systems Agency, Dec. 30 1996. http://spider.osfl.disa.mil/cm/how_to/howtoseg.pdf.

14. Lucent Technologies. Not So Bad Distribution (NSBD), 1998. http://www.bell-labs.com/project/nsbd/.

15. D. Mackenzie, R. McGrath, and N. Friedman. "Autoconf: Generating Automatic Configuration Scripts," Free Software Foundation, Inc, April 1994.

16. Marimba, Inc. "Castanet Product Family," 1998. http://www.marimba.com/datasheets/castanet-3_0-ds.html.

17. ObjectSpace, Inc. Voyager, 1998. http://www.objectspace.com.

18. Open Software Associates. OpenWEB netDeploy, 1998. http://www.osa.com.

19. Tivoli Systems. "Applications Management Specification," Version 2.0, Nov. 5 1997. http://www.tivoli.com/o_products/html/body_ams_spec.html.

20. Andrew Tridgell and Paul Mackerras. "The rsync algorithm," Technical Report TR-CS-96-05, June 1996. http://cs.anu.edu.au/techreports/1996/index.html.

21. E. Tryggeseth, B. Gulla, R. Conradi. "Modeling Systems with Variability using the PROTEUS Configuration Language," Proceedings of the 1995 International Symposium on System Configuration Management, Springer, 1995, pp. 216-240.

22. Twenty Twenty Software. PC-Install with Internet Extensions, 1998. http://www.twenty.com.