Agent-based Software Configuration and Deployment

by

Richard Scott Hall
B.S., University of Michigan, 1990
M.S., University of Colorado, 1993

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Doctor of Philosophy
Department of Computer Science
1999

This thesis entitled:
Agent-based Software Configuration and Deployment
written by Richard Scott Hall
has been approved for the Department of Computer Science

____
Alexander L. Wolf

____
Dennis Heimbigner

____
Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above-mentioned discipline.

Hall, Richard Scott (Ph.D., Computer Science)
Agent-based Software Configuration and Deployment
This thesis directed by Associate Professor Alexander L. Wolf

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 performed. It is necessary to introduce new software deployment technologies that leverage this connectivity. The Software Dock framework presented in this thesis creates 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 deployment schema for describing software systems, called the Deployable Software Description (DSD) format. The Software Dock also employs agents to traverse between software producers and consumers in order to perform software deployment activities by interpreting the deployment descriptions of the software systems. The Software Dock infrastructure allows software producers to offer to their customers high-level deployment services that were previously not possible.

Dedication

In memory of Jay H. Kegerreis, his impact will far exceed his lifetime.

Acknowledgements

The work in this thesis embodies a large effort on my part, but this effort would not have been possible without the help and support of many people. First, I thank my parents; this four-year dalliance was only possible because of them. I hope that one day I will give to my parents as much as they have given to me.

I thank my advisors, Alexander Wolf and Dennis Heimbigner, for giving me the opportunity to work on an interesting project and supporting my work and my ideas over the last four years. I am especially indebted to Dennis for his willingness and ability to listen and participate in my wacky discussions on whatever tickled my fancy. These experiences were the true definition of getting a doctorate to me.

I also thank the members of my research group, both past and present. In particular I acknowledge the contributions of Antonio Carzaniga, Andre van der Hoek, Mark Maybee, and Judy Stafford. These people were there for me throughout most of this ordeal and in some fashion aided me in getting through it. Despite all the aggravations and irritations along the way, we found plenty of ways to have fun.

Finally, I thank my committee members for the time and input and I thank anyone else I may have forgotten.

Table of Contents

Chapter 1 Introduction **1**

1.1 Motivation 4

1.1.1 Increased Software Complexity 4

1.1.2 Increased Software Producer and Consumer Connectivity 5

1.1.3 Continued and Growing Importance of Related Capabilities 6

1.1.3.1 Traditional Configuration Management 7

1.1.3.2 Software System Install, Update, and Remove 8

1.1.3.3 Software System and Site Description 9

1.1.3.4 Network Management and System Administration 9

1.1.3.5 Content Delivery 10

1.1.3.6 Configurable Distributed Systems 10

1.1.3.7 Component Models 11

1.2 Contributions 11

1.3 Outline of the Thesis 12

Chapter 2 Related Work **15**

2.1 Traditional Configuration Management 15

2.2 Software System Install, Update, and Remove 16

2.3 Software System and Site Description 18

2.4 Network Management and System Administration 20

2.5 Content Delivery 21

2.6 Configurable Distributed Systems 21

2.7 Component Models 24

Chapter 3 Software Deployment Life Cycle 26

3.1 Life Cycle Processes 27

3.1.1 Producer-side Processes 27

3.1.2 Consumer-side Processes 28

3.2 Life Cycle History 29

3.3 Life Cycle Process Characterization 31

Chapter 4 The Software Dock Architecture 33

Chapter 5 Software System Description 38

5.1 Definition of a Software System 38

5.2 Software System Description Issues 39

5.3 Requirements 41

5.3.1 Consumer Site Description Requirements 42

5.3.2 Software System Description Requirements 43

5.3.2.1 Assert constraints 43

5.3.2.2 Dependency constraints 44

5.3.2.3 Artifacts 46

5.3.2.4 Configuration 46

5.3.2.5 Activities 47

Chapter 6 The Deployable Software Description Format 49

6.1 Approach 49

6.2 Document Type Definition Overview 52

6.2.1 Family 52

6.2.2 Id 52

6.2.3 ImportedProperties 53

6.2.4 Properties and Property 53

6.2.5 Composition and CompositionRule 55

6.2.6 Assertions and Assertion 56

6.2.7 Dependencies and Dependency 57

6.2.8 Artifacts and Artifact 58

6.2.9 Interfaces and Interface 59

6.2.10 Notifications and Notification 60

6.2.11 Services and Service 60

6.2.12 Activities and Activity 61

6.3 Creating a Deployable Software Description Specification 62

Chapter 7 Software Dock Deployment Process Definitions 64

7.1 Generic Software Deployment Process Definition 64

7.2 Specific Software Deployment Process Definitions 66

7.2.1 Install Process 66

7.2.2 Update Process 67

7.2.3 Reconfigure Process 68

7.2.4 Adapt Process 68

7.2.5 Remove Process 69

Chapter 8 Out of Scope 70

8.1 Security and Privacy 70

8.2 Electronic Commerce 72

8.3 Error Recovery 72

8.4 Deployment Process Interactivity 73

Chapter 9 Evaluation 75

9.1 Deployment Solution Characterization and Capabilities 75

9.1.1 Abstractions 76

9.1.1.1 Consumer Abstraction 76

9.1.1.2 Software System Abstraction 77

9.1.1.3 Process Abstraction 78
9.1.2 Process Coverage 78
9.1.3 Coordination 79
9.1.4 General Requirements 79
9.2 Mapping the Software Dock to Requirements 82
9.3 Critical Analysis Of Related Technologies 86
9.3.1 Traditional Configuration Management 86
9.3.2 Software System Install, Update, and Remove 87
9.3.3 Software System and Site Description 89
9.3.3.1 The Open Software Description (OSD) Format 89
9.3.3.2 The Software Management Information Format (MIF) 92
9.3.3.3 Other Software System and Site Description Technologies 94
9.3.4 Network Management and System Administration 95
9.3.5 Content Delivery 98
9.3.6 Configurable Distributed Systems 98
9.3.7 Component Models 100
9.4 Usage Experiments 102
9.4.1 Online Learning Academy 102
9.4.1.1 Existing Deployment Solution 103
9.4.1.2 Software Dock Deployment Solution 104
9.4.2 The Chimera Open Hypermedia System 106
9.4.2.1 Existing Deployment Solution 107
9.4.2.2 Software Dock Deployment Solution 107
9.4.3 Java Development Kit 108
9.4.3.1 Existing Deployment Solution 108
9.4.3.2 Software Dock Deployment Solution 110

Chapter 10 Software Dock Prototype Implementation 114

10.1 Generic Dock 114

10.2 Events 115

10.3 Release Dock 119

10.4 The Field Dock 121

10.5 Agents 123

10.5.1 Agent Initialization 124

10.5.2 Event Dispatch and Process Activation 126

10.5.3 Assertion Processing 130

10.5.4 Dependency Processing 131

10.5.5 Activity and Interface Processing 132

10.5.6 Notification Processing 134

10.5.7 Artifact Processing 135

10.6 Schema Editor and Docking Station 136

Chapter 11 Future Research 139

Chapter 12 Conclusions 143

Appendix A Deployable Software Description Document Type Definition 150

Appendix B Deployable Software Description Example 153

Appendix C Open Software Description Example 159

Appendix D Management Information Format Example 160

Appendix E Software Dock/InstallShield Comparison Experiments 166

List of Tables

Table 1:	Software Dock comparison to InstallShield.	112
Table 2:	InstallShield deployment for the Java Development Kit.	167
Table 3:	Software Dock deployment for the Java Development Kit.	169
Table 4:	Software Dock deployment for the Java Development Kit on Solaris.	170

List of Figures

Figure 1: Relationship among software deployment related technologies. 7

Figure 2: A simple grammar for the consumer-side portion of the software deployment life cycle. 27

Figure 3: The Software Dock architecture. 34

Figure 4: The Configuration Editor. 126

Figure 5: The DSD Schema Editor. 137

Figure 6: The Software Dock Docking Station. 137

Chapter 1

Introduction

Interest in software deployment has recently intensified in both the research and commercial sectors. Software deployment itself is broad in scope and encompasses all of the activities that are performed after a software system is developed. The basic purpose of software deployment is to assemble a usable configuration of a given software system on a consumer's site, but it does not end here. Software deployment must also maintain the integrity of the deployed software system's configuration in the face of changes to its environment from both the producer and the consumer sides. These two participants, namely software producers and software consumers, motivate the commercial and research efforts in the field of software deployment. Therefore, any attempt to provide a solution for any piece of the software deployment problem space must take into account the competing and possibly contradicting desires of these two constituents.

Software producers have specific and separate concerns from the software consumer. In particular, software producers are concerned with software system packaging and release mechanisms that are simple, flexible, and extensible. After a software system is packaged and released the software producer must have an effective mechanism to advertise the release in order to notify interested consumers of its existence. There is little benefit for the software producer if its customers, both current and potential, are unaware of its products and services. Once a consumer has decided to deploy a software release on their site, the software producer needs reasonable access to the information available on that consumer's site in order to perform even the simplest deployment activities, such as computing platform and operating system information. Finally, because of efforts to speed software development, software producers are driven towards building systems of systems; that is, they build their system out of other software producers' systems. In this scenario, the software producer needs to maintain its autonomy with respect to other software producers so that it is not dependent upon their release schedules or release processes. All of these issues for the software producer are motivated by their primary concern to lower the total cost of support for their software systems.

On the other hand, the software consumer is concerned with making their software easier to maintain and manage. It is important to the software consumer to have a consistent, understandable mechanism for dealing with their installed software. Despite this required consistency, the software consumer still wants to maintain their autonomy with respect to their business practices and policies. In addition, privacy concerns result in the software consumer wanting to limit the type and scope of access that external entities, including software producers, have to their computing sites. The software consumer is not interested in macro-advertising techniques; rather they desire information that is of specific interest to them. All of these issues for the software consumer are motivated by their primary concern to lower the total cost of ownership for their software systems.

The reason for this apparent schism between software producers and consumers is largely due to the scope of the software deployment problem space. Software deployment lacks a formal definition and therefore this thesis presents a characterization of software deployment as a collection of interrelated processes that form a life cycle. This software deployment life cycle not only defines the scope of software deployment but also is a target for software deployment solutions.

Until recently, the notion of software deployment was limited to the installation and removal of a software system. The connectivity provided by large networks, such as the Internet, has affected how software deployment is viewed and performed. The simplistic view that software deployment merely provides a complete installation procedure for a software system on a CD-ROM is giving way to a more sophisticated process of ongoing cooperation and negotiation among software producers and consumers. This connectivity and cooperation allows software producers to offer their customers high-level deployment services that were previously not available. Already software system update has received wider support, but support for the other software deployment processes defined in this thesis, such as release, adapt, activate, deactivate, remove, and retire is still virtually non-existent. These processes are defined more fully in Chapter 3.

The vision of this thesis is to create a consistent, unified framework to support software deployment. This vision assumes that large networked environments, such as the Internet, are the medium of choice for distributing and deploying software systems. These networks are leveraged to provide semi-continuous, bi-directional communication between software producers and consumers. In this framework, a software system is not simply a collection of file artifacts, but also includes a description of the software system's essential deployment requirements.

In order to support such a vision, new enabling technologies must become available. These enabling technologies at a minimum must meet these requirements:

- 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 the automation of software deployment processes,
- provide a semantic model of target sites for deployment in order to describe the context in which deployment processes occur, and
- provide decentralized, flexible control for large numbers of software producers and consumers.

The Software Dock research project presented in this thesis serves as a foundation for research in fulfilling the vision of a consistent, unified software deployment framework. The Software Dock is a framework of loosely coupled, cooperating, distributed components. The Software Dock framework supports software producers by providing the "release dock" server that acts as a repository of software system releases. At the heart of the release dock is a standard schema for describing software systems. The "field dock" server supports the software 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. The agents perform their tasks by interpreting the descriptions of both the software system and the target consumer site descriptions. A wide-area event notification system connects release docks to field docks and enables asynchronous, bi-directional connectivity.

1.1 Motivation

Research into the field of software deployment is motivated by a variety of factors. The most important factors are discussed in the following subsections and include the increased complexity of software systems, the increased connectivity of software producers and consumers, and the continued and growing importance of related capabilities, such as configuration management and network administration.

1.1.1 Increased Software Complexity

The increased complexity of software systems has exacerbated the need for mature software deployment support. The complexity of software systems is increasing to the point where the cost of ownership is almost prohibitive for the software consumer. This growing complexity results from a variety of issues. First and foremost, software continues to evolve and the number of tasks that it performs continues to grow. Consumer demand, technological advances, and competitive advantage continue to fuel this growth.

Modern software development techniques are also responsible for the growing complexity of software systems. The push towards component-based technology is creating a situation where an increasing number of software systems are built by combining and coordinating other software systems; these types of systems are referred to as a system of systems. Software producers are put into a situation where they are no longer in control of all of the pieces that comprise their system. Required subsystems for any given software system are commonly developed and released by independent, autonomous organizations. This forces collaboration between organizations that have little opportunity or support for cooperation.

As networked environments, such as local area, intranet, and internet, continue to become the norm, software systems are developed to leverage this connectivity. A common and popular approach to create these new classes of distributed software systems extends the component-based approach with distributed capabilities. Technologies such as CORBA [42], DCOM [8], and Java RMI [53] create potentially unmanageable relationships and dependencies among subsystem components possibly over wide geographic areas. In such scenarios, there is little, if any support for the sophisticated software deployment tasks that are required.

1.1.2 Increased Software Producer and Consumer Connectivity

The growing connectivity between software producers and software consumers has given software producers a new tool to leverage in their attempts to offer their customers high-level deployment services. For example, the widespread popularity of the Internet is an impetus causing software producers to rethink their deployment activities. Software producers are enticed by this connectivity because they see the global market that it creates. In order to take advantage of this virtual marketplace, they must provide secure distribution, licensing, and billing for software systems and services; producers have largely ignored these issues in the past. From the perspective of the software consumer, they want to purchase, install, maintain, and support their software systems via the Internet. As a result it is becoming increasingly difficult for software producers to develop their software systems without taking these issues into consideration.

A byproduct of the connectivity offered by the Internet is the new interaction possibilities between software producers and consumers. Through this connectivity, producers can receive feedback during a software system's life cycle and respond to it. This requires that software deployment systems support an aspect of deployment that was not directly available in the past. The consumer's expectations, by having this level of connectivity, have increased and they are demanding a new level of quality of service from software producers. For this reason, it is imperative that deployment activities become an integral part of the software development process and, as such, have a full set of tools and infrastructure to support them.

1.1.3 Continued and Growing Importance of Related Capabilities

Capabilities and technologies, such as traditional configuration management, software system install, update, and remove, software system description, network management and system administration, and content delivery, have developed and matured independently of software deployment. These technologies are clearly related to the more general notion of software deployment [see Figure 1]. Other technologies, such as configurable distributed systems and component models have a relationship to deployment, although it is less direct than the previously mentioned technological areas. Of all of these related technologies, some have existed for many years, while others have come to the forefront more recently. The continued and growing importance of these related capabilities requires that they be examined in detail with respect to software deployment. The following subsections provide an overview of each of these areas, while more detailed information is presented in Chapter 2 and Section 9.3. At a minimum, software deployment must combine and leverage many of the accomplishments in these individual disciplines.

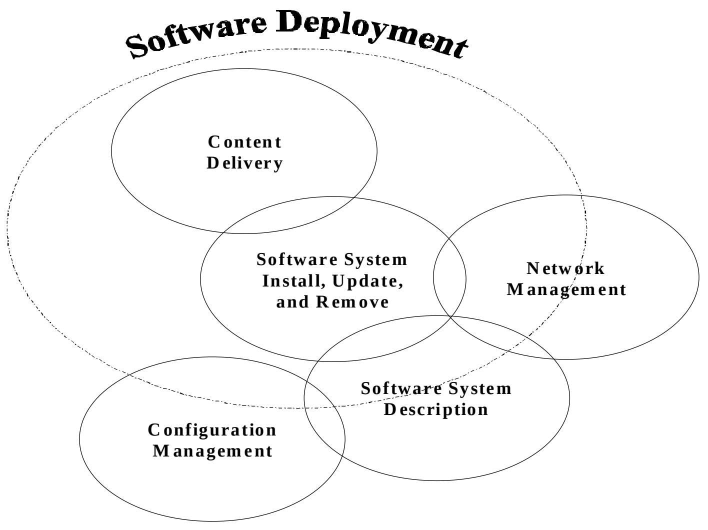


Figure 1: Relationship among software deployment related technologies.

1.1.3.1 Traditional Configuration Management

Configuration management technologies have existed for quite some time and their importance continues to grow. These technologies provide solutions for dealing with the software development activities of source code versioning and artifact derivation. The notion of artifact derivation is not directly applicable to software deployment nor explicitly is the notion of source code versioning, but configuration information and the system model captured by these technologies is relevant. Configuration management systems manage relationships that define consistent software artifact sets that comprise a given version of a software system. There are two types of versions in configuration management systems; a consistent set of time-related artifacts comprise a *revision*, while multiple disjoint, consistent artifact sets of a specific revision are called *variants*.

A large portion of configuration management technology centers on using the relationships among software artifacts to enable selection mechanisms that extract a consistent configuration of a given software system. These notions directly relate to software deployment because most software deployment processes require knowledge about consistent software system configurations in order to ensure successful deployment and operation.

1.1.3.2 Software System Install, Update, and Remove

Technologies that deal with the installation, update and removal of software systems are directly related to software deployment. In the past, software deployment effort focused on the installation process. Despite the fact that software systems have required installation procedures for nearly as long as software producers have been creating software systems, there is no complete or unified method for performing software installation. Many non-unified approaches exist to support software system install, where the main purpose of these technologies is to install an operational instance of a single version of a software system.

Software system update, which is the process of keeping a software system up-to-date with respect to chronological versioning, has recently started to receive attention. The growing popularity of networked environments, such as the Internet, has changed this focus. Producers of installation tools have begun to extend their installation systems to take advantage of networked environments, and some have come to an important observation; installation is a special case of update. The install process is an update where none of the software system's components are present on the target site. This realization has led to technologies that support both the install and update processes, as well as technologies that only support software system update.

Software system removal is a consequence of software system install and update. Despite its apparent obviousness, removal is not fully supported by current installation technologies. Numerous technologies that specialize in removing or uninstalling software systems currently exist in the commercial market. The sole purpose of these technologies is to remove all artifacts and to undo any effects that may have resulted from the installation of a given software system.

1.1.3.3 Software System and Site Description

Despite the fact that that there has been little effort to create a unified framework to support software deployment, there has been some effort to create standard description technologies for software systems. Standard description technologies are important since any effort to create a complete, unified software deployment technology requires some form of standard for describing the deployment details of a software system. Without such information it is not possible to understand the deployment requirements of a given software system nor is it possible to understand how to manipulate a given software system. Efforts to devise software, hardware, package, and release standards are driving many research and corporate organizations. The main purpose behind these standard description efforts is to create contexts about the elements they describe in order to simplify or automate the processing of those elements.

1.1.3.4 Network Management and System Administration

One of the first areas to realize and attempt to address the complexity of software deployment in a networked environment was network management and system administration. In the beginning, network management systems largely monitored and managed the hardware elements on a network. Network management has grown to include many of the software deployment activities defined in Chapter 3, such as install, activate, deactivate, update, and remove. Much of this growth into software deployment is because software producers largely ended their involvement in the software deployment life cycle after the consumer bought a software system. Producers of network management systems have concentrated most of their effort in creating centralized deployment systems. These systems gather all deployment-related knowledge about their software systems in a centralized database. This approach has resulted from the assumption that one organization is in control of both sides of the deployment equation, i.e., the producer side and the consumer side.

As the efforts in other areas of software deployment capabilities and activities continues to grow and become successful, network management will likely reduce its concentration on software deployment activities and focus more clearly on issues that fall outside of the software deployment life cycle. These areas include software system runtime management, software monitoring, load balancing, and error recovery.

1.1.3.5 Content Delivery

The growing popularity of networked environments, such as the Internet, has created a new niche of software systems that leverage the connectivity and make the delivery of content itself a primary focus. The perceived benefit of content delivery systems is the efficient, timely delivery of content to interested parties. Some content delivery technologies directly address software deployment issues by using their techniques to update installed software systems.

Content delivery is a generic term referring to the movement of any type of artifact or artifacts from one point on the network to another. Given such a broad definition it is clear that techniques used in content delivery systems are required by or affect many of the software deployment processes and activities. As such, it is possible for software deployment technologies to leverage the success in these areas in order to perform some of their tasks. At the very least, deployment technologies must introduce related content delivery systems that are better suited for their tasks if the current systems do not address their needs.

1.1.3.6 Configurable Distributed Systems

Research in the area of configurable distributed systems (CDS) examines the special requirements of software systems that run continuously or for very long periods of time. One specific area of interest in the CDS community is that of runtime reconfiguration. For software systems that run continuously, it is not possible to stop system operation in order to perform configuration changes or updates. Although software deployment does not specifically address runtime reconfiguration, it is possible that techniques or approaches used in CDS research may be applicable to software deployment.

1.1.3.7 Component Models

One of the contributing factors to the growing complexity of software deployment is that of component-based development. The component-based approach centers on the creation of standard models for creating software modules called components. It is possible for high-level programming tools to manipulate components using the standard interfaces or properties defined by the component model; the goal is to create a programming experience that is akin to wiring reusable components together rather than writing specific code. Component models are not within the purview of software deployment, but the use of component-based tools to define a software system configuration by connecting and packaging the required components is applicable.

1.2 Contributions

This thesis makes a number of contributions to the area of software deployment. The primary contribution is a standard schema for describing software systems for deployment. The intent of this standard schema is to provide enough information about the deployment requirements of a software system to enable an automated or greatly simplified approach to its software deployment activities. Any attempt to create unified support requires some mechanism to capture deployment-related characteristics of a software system; the standard schema presented in this thesis is a step to realizing this goal.

The *software deployment life cycle* is a contribution for defining and understanding the scope of the software deployment problem space. The software deployment life cycle is important because it not only defines the scope of the software deployment problem space, but it also relates many activities and processes that were previously not considered part of an overall process. Each activity in the software deployment life cycle has a specific role in the issues surrounding software deployment research. The life cycle is considered an evolving target, however. As a result, it can grow to include new or unforeseen deployment processes.

Another important contribution of this thesis is the definition of a generic algorithm for performing most software deployment processes. The generic deployment process algorithm resulted from implementing processes to interpret the standard schema defined in this thesis. Due to the nature of the standard schema implementation, which defines a software system in terms of properties that form consistent configurations of the software system, an abstract process algorithm was uncovered and refined. As a result, it was shown that most of the deployment processes have a great deal of commonality.

The architecture and implementation of a prototypical software deployment framework, called the Software Dock, provide additional contributions to the field of software deployment. The prototype explores the effectiveness of an agent-based and event-based approach to supporting software deployment in a unified, consistent manner. The architecture of the prototype has exhibited great flexibility and effectiveness and it has provided valuable lessons and insights throughout its implementation.

1.3 Outline of the Thesis

The approach of this thesis was based on the realization that many traditional and emerging technologies were addressing pieces of a larger issue, namely software deployment. These related technologies and their approaches are discussed in Chapter 2. Despite the continued and growing importance of these related technologies, software deployment was not addressed explicitly. This lack of focus resulted from the lack of a clear definition or understanding of software deployment. To address this issue, a characterization of software deployment, referred to as the software deployment life cycle, was created and is presented in Chapter 3. This evolving characterization defines the scope of the software deployment problem space.

Given a more focused definition of software deployment, an architectural solution, called the Software Dock, was devised to create a unified software deployment framework; the Software Dock architecture is presented in Chapter 4. A critical aspect of this framework was its reliance on declarative specifications for performing software deployment processes. A large portion of this thesis is dedicated to the effort that has gone into devising a standard schema to describe software systems for deployment. Initially, the abstract notion of software system description was examined and a set of description requirements was formed; the results of this process are presented in Chapter 5.

After the software system description requirements were formalized, the definition of a standard schema to address many of these requirements was formulated. The Deployable Software Description format was the result of this process and it is presented in Chapter 6. This standard schema captures the deployment requirements of a software system with the intention of automating its deployment processes through the use of the Software Dock's collection of generic deployment processes. Agents are used in the Software Dock framework to realize these generic deployment processes; they do so by interpreting the descriptions of each software system. The creation of the deployment agents resulted in the discovery of an abstract deployment algorithm as well as the specific process definitions for each deployment task; the result of this work is presented in Chapter 7.

In order to make this thesis tenable, it was necessary to limit the scope of the proposed solution. The critical areas that were deemed out of scope, namely security, electronic commerce, error recovery, and deployment process interactivity, are discussed in Chapter 8. The evaluation of the approach described by this thesis is presented in Chapter 9. The evaluation addresses the characterization and criteria of software deployment solutions in general and then applies them to the Software Dock framework. Related technologies are critically evaluated in order to illustrate that there is no current approach that address software deployment in a unified manner. Further, three usage experiments are described that illustrate the effectiveness of the presented solution.

Details of the prototype Software Dock implementation are presented in Chapter 10. Future research is presented in Chapter 11, which details the direction of the continued work in this area, followed by the conclusions in Chapter 12.

Chapter 2

Related Work

Since the scope of the Software Dock project is so large, there are many related technologies. This section attempts to cover a representative sample of the most important related work. For critical evaluation of these related technologies refer to Section 9.3.

2.1 Traditional Configuration Management

Traditional configuration management systems are important to software deployment because they employ semantic models to describe software systems and they introduce the notion of configuration selection. Specifically, traditional configuration management systems have introduced software system modeling approaches to deal with the versioning of software systems. Common configuration management tools such as RCS [59] and SCCS [48] have introduced somewhat simplistic software system models based on source code-level differential analysis. In such models a software system is defined as a set of source code revisions that define a consistent configuration or baseline of a given software system. These tools then enable the user to select consistent configurations from its version space.

Other configuration management technologies have introduced more sophisticated software system modeling languages, such as Adele [16] and PCL [63]. Adele provides a system modeling language to describe software system configurations. In order to describe software system configurations, Adele introduces the notions of interfaces and realizations of those interfaces. A specific interface may have many revisions, where each revision may have multiple realizations and these realizations may also have many revisions. The notion of a family is introduced to refer to the set of all versions of an interface and their realizations. In Adele, it is possible to annotate the components of a software family with arbitrary attributes that denote specific semantic properties. The main contribution of Adele is the use of constraints over these attributes to express the composition of valid configurations. Adele is able to select valid software system configurations when presented with specific attribute values by resolving the associated constraints.

PCL is a language for software system modeling, configuration definition, and system building based on the family concept. The family description encompasses all potential variability of a system and may represent any kind of entity. A family is organized as a layered composition structure, which describes a single logical entity that may contain or be contained in other entities. A family definition includes attributes, *parts* or sub-components, relationships, and physical objects (e.g., files). The family attributes are used to characterize potential variability. Using if-then-else expressions over the attribute values throughout the family description supports structural variability. Inheritance is supported when defining subsequent, related families. Other specific language constructs in PCL, such as a *tool*, support the building process for a software system.

2.2 Software System Install, Update, and Remove

Software system install, update, and remove systems are important to software deployment because they are all necessary elements of software deployment. A host of installation tools exist in the commercial world, such as InstallShield [27], PC-Install [64], and WISE [66]. A common approach used by these systems is to generate an installation script that is then executed by a proprietary interpreter. The creation of the installation script is not necessarily a programming exercise; a tool with a graphical user interface may generate the installation script by querying the user for common information. Direct editing of the installation script is necessary to support non-standard installations. The software release is generally modeled as groups of related file artifacts and consumer site state changes. These approaches provide support for selecting a valid configuration of a software release. Standard access to the underlying computing site is also provided; this access might include platform specific state inspection and manipulation, such as access to the Registry under Microsoft Windows 95™, or access to the file system. Many of these installation tools also provide a mechanism for removing the installed software release. To facilitate Web distribution, some of these tools can create self-contained installation executables where the artifacts, the install script, and the script interpreter are bundled into a self-extracting archive.

Recent extensions to some installation tools have included support for the connectivity of the Internet. An approach used by InstallFromTheWeb [28] and PC-Install with Internet Extensions [65] is to use the plug-in technology of modern Web browsers to perform installs directly from a Web page. The InstallFromTheWeb approach merely wraps an existing installation program with additional information to automate the download and execution process, while PC-Install with Internet Extensions uses its plug-in technology to embed a traditional installation script interpreter in the Web browser.

More sophisticated approaches, such as netDeploy [44], have also included support for the update process. Support for installation in netDeploy is similar to the other installation tools, but it has added a Web browser helper tool, called the *Launcher*, that verifies whether the deployed software release is up-to-date with respect to the producer site. Each time that an application deployed by netDeploy is executed any discovered updates are downloaded to the consumer site.

Other approaches exist specifically to simplify the process of updating deployed software releases. Update tools, such as OilChange [36], LiveUpdate Pro [56], and TuneUp.com [57], use central databases to maintain information on thousands of applications, utilities, and drivers. A client utility scans the consumer's site for software releases and then determines whether there are updates for any recognized software from the central database. If any outdated software is discovered, the client utility can help the user retrieve and possibly even install the updates.

Despite the reasonable support the above tools provide for installation and update, the support they provide for software release removal is insufficient. A host of commercial utilities, called uninstallers, exist for the Microsoft Windows 95 platform to aid in effective and complete removal of software systems [37][55]. These tools examine a consumer’s site to find unused dynamically loaded libraries and other remnants of partially removed software systems in order to remove them completely.

2.3 Software System and Site Description

Any attempt to provide unified support for software deployment requires some notion of software system description to capture deployment knowledge about a software system. The Open Software Description (OSD) [25] format is a W3 Consortium proposed standard created jointly by Microsoft and Marimba. OSD provides a vocabulary for packaging software; this includes describing software components, their versions, underlying structure, and relationships among components. OSD, which is one piece of Microsoft's Zero Administration Initiative [39], is related to Microsoft's Channel Definition Format (CDF) [15] for “push” content. The intent is to use the two standards in conjunction with “push” technologies to enable software systems to automatically install and update themselves. The syntax for both OSD and CDF are based on the Extensible Markup Language (XML) [7].

The Desktop Management Task Force (DMTF) is an industry consortium chartered with the development, support, and maintenance of management standards for personal computer systems and products. An initial result of the DMTF effort is the Desktop Management Interface (DMI) [11], which creates a common interface layer to access management information on computing systems. An effort related to DMI is the Management Information Format (MIF) [14], which is a common, hierarchical data model used in describing all aspects of computing systems, including software systems. A major contribution by DMTF is the formation of working groups to create standard MIF schema to describe various aspects of computing systems. The focus of the discussion here revolves around the Software MIF [13] created by the Application Management working group. Currently, DMTF is moving towards a new object-oriented data model that is called the Common Information Model (CIM) [12]. DMTF is currently mapping the Software MIF to this model.

A superset to the Software MIF was created by Tivoli and is called the Application Management Specification (AMS) [61]; therefore, the discussion here is also relevant to the Software MIF. 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 describes a static configuration of a software system that is installed and monitored at a consumer site. AMS is intended for use in an enterprise framework, such as Tivoli Enterprise [60], where there is a central administration authority that is responsible for maintaining the state of deployed software systems.

The Defense Information Infrastructure Common Operating Environment (DII COE) [29] is a Department of Defense effort to restrict the set of components used to build their software systems. The DII COE supports, among other things, a standard means for packaging components for delivery and installation. These packages are called *segments* [30], 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. A segment is a static depiction of a deployed software system configuration. The support provided is intended for a central administration authority.

Additional approaches, such as GNU Autoconf [33], try to resolve consumer site description by using scripts and heuristics to directly examine the state of a site. The results of the direct examination are used to derive or instantiate deployment routines that are parameterized by the consumer site property values. The Registry [26] is a hierarchical registry of consumer site information for the Microsoft Windows 95 platform. The schema used in this registry is only partially standardized but it does contain much information that is needed for deployment activities on the Windows platform.

The Redhat Package Manager (RPM) [4] is a tool for the Linux user community that provides many software deployment features. RPM is also in installation utility, but since RPM's approach revolves around descriptions of software releases it is included here. RPM packages contain the software system artifacts and a description of the software system; this description includes constraints, dependencies, artifacts, and activities in the form of scripts. The granularity of a RPM package is a single revision and a single variant. Some forms of configuration selection are supported. Several support tools for retrieving and installing software packages from a network accompany RPM.

2.4 Network Management and System Administration

Network management and system administration systems exist in the commercial world to help enterprises manage the hardware and software on a network of computers. They are important to software deployment because they attempt to address many of the same issues of the software deployment life cycle. A representative sample of the most important of these systems includes Tivoli Enterprise [60], HP OpenView [23], and Microsoft System Management Server (SMS) [40]. All of these systems follow a very similar approach; they provide integrated suites of management tools built on top of a network management framework. These systems each provide their own model of software systems and consumer sites. These models are stored in centralized databases along with site inventories gathered from their managed sites. The management tools use this information to perform software deployment processes, such as software system install, update, and remove.

Facilities are also provided to control, manage, and limit the operations performed on managed sites in order to create consistency among sites. These systems extend into the runtime environment as well, with mechanisms to monitor software systems and send notifications in response to certain conditions.

2.5 Content Delivery

In the end, most software deployment activities either reduce to or require the transfer of software artifacts. A class of commercial and research tools exist to support artifact delivery and update; some of these systems include PointCast [47], Castanet [35], NSBD [32], rsync [62], and Minstrel [22]. In most of these systems, the main value is simply the transfer of the artifacts. Recently Castanet has started to reposition itself as an application server where an application is defined as a set of file and directory artifacts and specific changes to the consumer site, such as entries in the Registry on the Microsoft Window 95 platform. In general, these tools assume very little about the content that they deliver and simply treat it as a collection of file and directory artifacts; an exception is PointCast, which displays its content with a viewer. The specific approach followed by each individual tool varies and includes content mirroring, consumer-side polling, and server-side publish-and-subscribe techniques.

2.6 Configurable Distributed Systems

Software systems that run continuously or for very long periods of time pose unique problems when they are reconfigured or updated. The primary focus of research in the field of configurable distributed systems (CDS) is that of runtime reconfiguration and update. More precisely this involves the addition, removal, or replacement of software system components without ceasing or seriously disrupting the operation of an executing software system. The issues of runtime reconfiguration go beyond the scope of software deployment as defined in this thesis [see Chapter 3]. The level of support for runtime services in a deployment framework is an open debate and it is unclear where and if a precise dividing line exists. Despite this, it is still important to understand the approach of the CDS community in resolving these reconfiguration issues in order to understand the difference between CDS and deployment and to leverage ideas when applicable.

CDS research has followed a common approach to resolve runtime configuration. This common approach introduces a reconfiguration layer or environment to an ordinary distributed system. The degree of intrusiveness of the reconfiguration layer varies according to the specific research approach, but the intrusiveness imposes certain limitations or standards on a software system. The reconfiguration layer leverages the knowledge of these standards in order to perform runtime reconfiguration; specific research approaches are described in the later paragraphs of this section. Regardless of the degree of intrusiveness, the goal is to enable dynamic reconfiguration of a distributed software system at runtime with minimal disruption to its normal operation.

The trend in CDS research is to use a high-level software system description to enable runtime changes. The software system description specifies the components that comprise a given software system and the connections among those components. The connections are specific types of allowable communications channels for the given runtime environment, or they are standard middleware such as CORBA [42]. The main thrust of runtime reconfiguration is to "passivate" or suspend the smallest number of software system components in order to enact a reconfiguration. Changes to the software system description are then reflected in the running system using the tools of the runtime environment. The main complication in this process is maintaining the integrity of the software system while the changes occur. Once the reconfiguration is complete, all components are activated and the operation of the system continues with the new configuration.

Some approaches [1][6][17][34][45][46] with a higher degree of intrusiveness require a software system to subscribe to a common component model or architectural style. In these research projects, a component must either be derived from a common base component or it must implement a specific interface or set of interfaces. In the case of common base component derivatives, it is possible for a runtime reconfiguration environment to effectively wrap an application component with runtime reconfiguration capabilities. In the case of common interfaces, the runtime reconfiguration environment places specific semantics on the standard interfaces, and therefore understands how to manipulate a component in the face of runtime reconfiguration. In this latter approach, it is the responsibility of the software system developer to make sure that the implementation of the standard interfaces conforms to the proper semantics.

One particular approach [5] proposes an agent extension to a more traditional CDS system called PRISMA [67]. Agents are used to improve efficiency by moving to the site where a distributed component is executing, from this close proximity the agent carries out a CDS reconfigure or update task by passivating and rebinding the necessary components. This approach further proposes using the agents to actually deliver the new component code.

An approach presented in [38] uses proxy references to server components instead of direct references. In this approach the intrusiveness of the reconfiguration layer only corresponds to the use of proxy references, while the actual structure of the software system is not dictated. Through the use of these proxy references, this work demonstrates that it is possible to account for some forms of server component failure, movement, and replacement. The end result is minimal operating disruption for clients of the server component.

Some approaches minimize their intrusiveness by depending on common middleware, such as CORBA. Even though the use of CORBA is a limitation imposed on a software system, this limitation is arguably less intrusive than systems that impose non-standard component models since CORBA is commonly used. One such research project [50] uses a transactional, rule-based workflow environment to perform dynamic reconfiguration. A workflow script describes component tasks of a software system and the temporal and dataflow dependencies among the tasks. The workflow script is instantiated by the workflow environment to create a running configuration of the software system. Task controllers are associated with each task in the runtime environment to correctly handle the temporal and data dependencies among the tasks and to manage runtime reconfiguration directives.

Another CDS research project [58] uses a prescriptive language to model valid configurations of software systems and computing sites in a generic fashion. This system models physical artifacts, such as directories and files, by modeling the properties of the artifacts. The modeled artifacts are comprised of methods to "get" and "set" the associated properties of the artifact. The prescription language is used to create logical expressions over the modeled artifacts. A valid configuration of a software system, for example, is modeled as the set of all logical expressions that sufficiently describe the desired valid configuration. From the configuration prescription, it is possible to automatically verify and repair the configuration of an installed software system by "getting" and "setting" the properties of the modeled artifacts. This system in particular is different than most CDS systems because it does not deal primarily with runtime issues.

2.7 Component Models

Component models such as JavaBeans [51], Enterprise JavaBeans [52], ActiveX [41], and the Common Object Model (COM) [49] are intended to facilitate interoperability among reusable modules of functionality. Interoperability is fostered by the introduction of standard interfaces and stylistic programming practices. These component models are intrusive with respect to the development of a software system because they require that the software system be implemented according to the defined component framework, although it is possible to create component model wrappers for legacy software systems. The result of this approach is that high-level programming tools can manipulate components in such a fashion that allows a programmer to build a software system by merely connecting components to one another. The description of a software system that is defined by connecting components together could be viewed as a configuration description of the software system. It is possible that this configuration description could be used useful for deployment purposes, since, at the very least, it serves as a bill of materials for the components that comprise the software system.

Examining the specific component models more closely is necessary to reveal relationships to deployment. In the JavaBean approach, a JavaBean is specifically described as a reusable software component that can be manipulated visually by a builder tool. A builder tool may be a web page builder, visual application builder, or a GUI layout builder. JavaBeans support runtime type information, appearance and behavior customization, event communication, properties, and persistence. JavaBeans are primarily intended for single-address space software systems. Enterprise JavaBeans extend the JavaBean model to include the notion of servers and clients, thus, supporting multi-address space software systems. Whereas JavaBeans are intended for manipulation with visual builder tools, Enterprise JavaBeans do not lend themselves as easily to visual manipulation due to their complexity. Despite the higher complexity, both JavaBeans and Enterprise JavaBeans are intended to build systems by connecting components to one another, thus defining a software system configuration. Additionally, both JavaBeans and Enterprise JavaBeans define standard packaging methods using Java Archive (JAR) files. JAR files are deployable artifacts that contain all of the necessary Java classes for a particular component. The JAR files themselves also include a manifest file that provides some declarative information about the components contained in the JAR file.

The ActiveX technology from Microsoft is very similar in intent and purpose to JavaBeans. ActiveX is built on top of a component model called Common Object Model (COM) and also relies on the Object Linking and Embedding (OLE) [41][49] services in the Microsoft Windows operating system. A distributed version of COM, called Distributed COM (DCOM) [8], extends the ActiveX component model to networked environments. An ActiveX control exposes properties and events that it exports and allows manipulation through property pages, much like JavaBeans.

Chapter 3

Software Deployment Life Cycle

In the past, software deployment was largely defined as the installation of a software system; this view of software deployment is simplistic and incomplete. Software deployment is actually a collection of interrelated activities that form the *software deployment life cycle*. The software deployment life cycle, as defined in this thesis, is an evolving collection of processes that include release, retire, install, activate, deactivate, reconfigure, update, adapt, and remove.

The relationships that exist among the processes are complex. Figure 2 presents a simple grammar to describe the relationships between the consumer-side deployment processes. This simple grammar is used to create any valid consumer-side software life cycle sequence. For a given software system, the grammar does not depict a complete life cycle since it does not include producer-side processes. The inclusion of producer-side processes makes the resulting grammar overly complicated. This complexity occurs because parallelism exists between successive installs of a software system and also because the notion of retiring a software system prohibits the on-going operation of certain consumer-side processes, such as install.

Each life cycle process is described in more detail in the following subsection, followed by a brief history of the creation of the life cycle. Since the software deployment life cycle is an evolving definition, the final subsection describes a mechanism for characterizing processes to determine whether they warrant inclusion in the life cycle.

```
start        → install
install      → "install" activate | "install" configure | "install" remove
activate     → "activate" deactivate [ | "activate" configure]
deactivate   → "deactivate" configure | "deactivate" remove |
               "deactivate" activate
configure    → "configure" reconfigure | "configure" update |
               "configure" adapt
reconfigure  → "reconfigure" configure | "reconfigure" activate |
               "reconfigure" deactivate
update       → "update" configure | "update" activate |
               "update" deactivate
adapt        → "adapt" configure | "adapt" activate |
               "adapt" deactivate
remove       → "remove"
```

Figure 2: A simple grammar for the consumer-side portion of the software deployment life cycle.

3.1 Life Cycle Processes

The processes of the software deployment life cycle are performed on either the software producer or consumer side; the processes for each side are described below.

3.1.1 Producer-side Processes

The producer-side of the life cycle consists of two processes, ***release*** and ***retire***. The release process is the bridge between development and deployment. It encompasses all of 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, but also contains a description of the deployment requirements for the software system. 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 is distinct from 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 system 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 withdrawn.

3.1.2 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 complete, 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 making a deployed software system executable or usable. For a simple tool, activation involves establishing some form of command (or click-able graphical icon) for executing the binary component of the tool. For a distributed system, it is possible that multiple components need to execute in parallel, such as server and client components. 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. Update deploys a new, previously unavailable configuration of a software system. An update becomes necessary when a software producer makes a change to the description of a deployed software system. The changes to the software system's description may denote a new version, a content update, or simply a description update.

The reconfigure process, like install, also modifies a previously installed software system, but its purpose is to select a different configuration of a deployed software system from its existing 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 deployed software system. Adaptation becomes necessary when a change is made to the local consumer site that affects the deployed software system. For example, when a required software system file is deleted or corrupted, the adapt process determines the problematic file and replaces it.

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 were caused by previous deployment activities for a given software system. The remove process must pay special attention 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 system in such a way as to not violate these dependencies and constraints.

3.2 Life Cycle History

The software deployment life cycle was created through a refinement process that started with real-world and research experience in deploying software systems. The Software Dock project was an outgrowth of another research project called the Software Release Manager (SRM) [24]. The main focus of SRM was the activity of software release. SRM created a repository where software producers could deposit their software release packages for subsequent retrieval by consumers. A typical software release package in SRM was simply a file artifact archive. The primary contribution SRM offered to the area of software release was the inclusion of a dependency notion among software releases. When a software release archive was submitted to SRM the dependencies on specific versions of other software releases were also entered. The end result was that consumers retrieving software releases through SRM could automatically retrieve all dependent software releases that were required to make the desired software release operational.

SRM did not, however, provide any support for performing any further deployment processes, such as install. The notion of the Software Dock was created as a deployment system that not only supported the release activity, but also supported follow-on deployment activities. As a result of this lineage, the software deployment life cycle in its infancy included the release, install, update, remove, and retire processes. Of these processes, only the update process does not appear to follow directly from the SRM experience. The update process was included, though, as a direct result of what SRM was trying to manage; the tangled dependencies between updated versions of software releases.

Originally, the update process had a larger scope than its current definition within the life cycle. The notion of an update included responses to producer-side events, such as the release of a new version of a software system, and consumer-side events, such as the deletion of a file. After further experimentation with early prototypes of the Software Dock, it was clear that two deployment processes were necessary to maintain cohesiveness. Therefore, the original update process was divided into two process, update and adapt. In this refinement, the update process was only related to producer-side events and the adapt process was only related to consumer-side events.

Continued experimentation led to the realization that another deployment process was necessary; an examination of the consumer-side deployment processes up until this point reveals the missing process. First, the install process selects and installs a configuration of a software release on the consumer site. The update process updates a configuration of a software release with respect to a new configuration from the producer site. The adapt process ensures that the current configuration of a software release remains valid at the consumer site. Finally, the remove process removes a software configuration from the consumer site. None of these processes enables the consumer to simply change the initially installed configuration of a software release. As a result, it was necessary to define the reconfigure process. These five processes, install, update, adapt, reconfigure, and remove, allow the consumer to completely manage the software release configurations at their site.

The activate and deactivate processes were added to the life cycle after serious consideration and debate. These deployment activities deserve special consideration since they border on the area of application frameworks. It is unclear what level of support a deployment framework can provide for activation and deactivation and still remain within the scope of the deployment problem space. There is no question, for example, that placing an icon used to execute a software system on the desktop of a graphical user environment is in the scope of software deployment. This process is directly analogous to placing an activation routine on the desktop, thus it appears that making a software system usable is within the domain of software deployment. In the end, proper deployment of a software system is of no value to software consumers if they are unable to start the deployed software system. The activation and deactivation processes require consideration within the scope of software deployment, but this consideration must be limited.

3.3 Life Cycle Process Characterization

The software deployment life cycle stated in this thesis is, by default, not complete since it is characterized as an evolving definition. As such, it is important to characterize the processes that belong in the life cycle in order to create a guide for adding new processes to it. In order to include a new process in the software deployment life cycle, the new process must perform one of the following functions and attempt to do so in a minimal number of steps:

- create or remove valid configurations from the set of all valid configurations of a software release,
- transform one valid configuration to another valid configuration or an invalid configuration to a valid configuration of a software release, or
- enable the usage of a configuration of a software release.

The first part of the characterization captures producer-side processes such as release and retire. The second part captures consumer-side processes that manipulate the configuration of a software release; the absence or empty configuration of a software release is considered a valid configuration. The last part of the characterization captures the consumer-side processes of activating and deactivating a software release configuration. If a new process performs functionality in one of these three areas and that functionality is not available from any of the existing processes, then that new process is a candidate for addition to the life cycle.

The notion that the deployment processes must attempt to perform their function in a minimal number of steps is important. From an abstract point of view, the update, adapt, and reconfigure processes are all simply instances of a remove followed by an install. Thus, update, adapt, and reconfigure seem superfluous. For the sake of efficiency, however, this is not true. A remove followed by an install always performs the maximum number of steps necessary to perform the desired functionality of the individual deployment processes. For example, a reconfigure does not require the removal and then an entire reinstall of a software release configuration and its dependent subsystems in order to perform the reconfiguration. Although some reconfigure processes may have this result, most reconfigurations only require the removal and installation of a subset of the software configuration. The time to complete the reconfiguration operation in the former case is significant compared to the latter case, especially when the operation is performed on large software releases over wide-area networks.

Chapter 4

The Software Dock Architecture

The Software Dock research project, originally described in [21], addresses support for software deployment processes by creating a framework that enables cooperation among software producers themselves and between software producers and software consumers. The Software Dock architecture [see Figure 3] defines components that represent these two main participants in the software deployment problem space. The *release dock* represents the software producer and the *field dock* represents the software consumer. A third component, called the *interdock*, also represents the software consumer by providing a global view if the consumer is part of a larger organization. In addition to these components the Software Dock employs *agents* to perform specific deployment process functionality and a *wide-area event system* to provide connectivity between the release docks and the field docks.

In the Software Dock architecture, 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 described using a standard deployment schema; the details of standard schema description for software systems are presented in

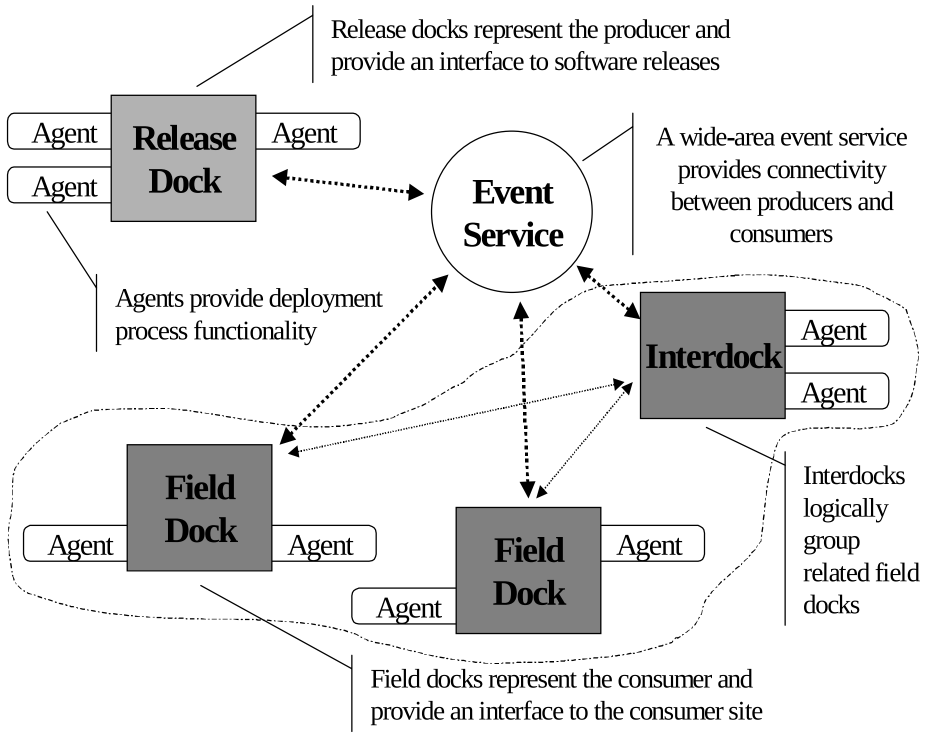

Release docks represent the producer and provide an interface to software releases
Agent
Release Dock
Agent
Agent
Event Service
A wide-area event service provides connectivity between producers and consumers
Agents provide deployment process functionality
Interdock
Agent
Agent
Field Dock
Agent
Agent
Field Dock
Agent
Agent
Interdocks logically group related field docks
Field docks represent the consumer and provide an interface to the consumer site

Figure 3: The Software Dock architecture.

Chapter 5 and Chapter 6. Each software release is accompanied with generic agents that perform software deployment processes by interpreting the 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 manages. Agents associated with deployed software systems can subscribe for these events to receive notifications about specific release-side occurrences, such as the release of 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 are 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 available to an agent at 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 interdock is a server that resides within a software consumer organization. The interdock has multiple purposes. It is a common repository of global information. This global information is used to logically group related field docks for deployment purposes. For example, an administrator may desire to install a given software system on a specific domain of field docks. Additional global information is stored in the interdock to create a global view of the organization for software deployment activities that span multiple field docks. For example, when installing a client software system it is necessary to determine if the required server software system exists within the consumer organization. Finally, the interdock may act as a halfway house for software releases. In such a scenario, software releases are brought in from a release dock and stored in the interdock. When deployment processes are performed they first check the interdock for their software release before contacting their release dock.

The release dock, field dock, and interdock are all very similar components. All are servers where agents can "dock" and perform activities. All manage a standardized, hierarchical registry of information that records the configuration or the contents of their respective sites and creates a common namespace within the framework. The registry model used in all of them is that of nested collections of attribute-value pairs, where the nested collections form a hierarchy. Any change to a registry generates an event that agents may receive in order to perform subsequent activities. The registry of the release dock largely provides a list of available software releases, whereas the registries of the field dock and interdock perform the valuable role of providing access to consumer-side information.

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 particular consumer site, while the interdock registry describes a global view of the consumer organization. By standardizing the information available within a consumer organization, the field dock and interdock create a common software deployment namespace for accessing consumer-side properties, such as operating system, computing platform information, and computing services. This information, when combined with the description of a software system, is used to perform specific software deployment processes.

Agents implement the actual software deployment process functionality. When the installation of a software system is requested on a given consumer site, initially only an agent responsible for installing the specific software system and the 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 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 has configured the software system for the specific target consumer site, it requests from its release dock the precise set of artifacts that correspond to the software system configuration.

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 information of the software system description and the consumer site configuration.

The wide-area event service [9] 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 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.

Chapter 5

Software System Description

This section discusses issues that are related to the general notion of software system description. A working definition of a software system is first presented in order to provide the context for the discussion of software system description. Issues relating to software system description are then presented. Finally, a set of requirements is defined for the description of software systems.

5.1 Definition of a Software System

In order to discuss the details of describing a software system, it is important to provide a more precise definition of a software system. The definition presented here only pertains to the view of a software system with respect to software deployment, and is not intended as a general definition. With this in mind, an initial definition is nothing more than the collection of artifacts that comprise a given software system. These artifacts may include executables, data, documentation, libraries, or anything else that a file may contain.

It is necessary to extend this initial definition because it is clearly not sufficient for anything other than the most basic instances of a software system. In particular, it makes the implicit assumption that a software system is monolithic. Invariably, the artifacts that comprise a software system do not exhibit an all-or-nothing relationship. For example, the collection of artifacts that comprise a software system usually varies with respect to the properties of platform, revision, or configuration. These variations are implicit relationships among the artifacts with respect to specific software system properties and require explicit description.

Some of these properties denote internal system relationships, such as revisions or configurations. Other properties denote external system relationships, such as consumer site or resource dependencies. In either case, it is necessary to describe the affect that each property has on the set of artifacts that comprise a given configuration of a software system. When these relationships are understood it is then possible to select the artifacts to deploy to a given consumer site by determining the values of the properties that affect the artifact set.

A further extension to this definition is necessary to capture any subsequent changes to the state of the target consumer site. A software system is not merely a collection of artifacts and their relationships, it may also include changes to the state of the target consumer site that cannot be represented as a file. These state changes may include information, such as the addition of an operating constraint, or modifications to the user environment, such as adding an icon to the desktop. The inclusion of consumer site state changes is particularly vague in order to include the notion of adding more abstract state changes, such as a software system adding interfaces and notifications to the consumer site.

Therefore, to rephrase the definition of a software system more concisely, a software system is a collection of artifacts, the relationships among artifacts and the external environment, and a collection of state changes to the target deployment site. A term that is used in conjunction with this definition throughout this thesis is software *family*. A software family is defined as the collection of all revisions and variants of a given software system.

5.2 Software System Description Issues

Software system description for deployment is complicated by many factors. In the past, software deployment was largely considered one process, installation. Recent definitions of software deployment have expanded to include the notion of the update process, but this is only now becoming the norm. An evolving definition of software deployment is introduced in this thesis that refers to software deployment as a collection of interrelated processes called the software deployment life cycle. These processes are release, install, update, reconfigure, adapt, activate, deactivate, remove, and retire as described in Chapter 3. In order to provide support for software deployment, a language or schema must provide enough information to support most, if not all, of these deployment processes.

Large network environments, such as the Internet, further complicate support for the software deployment life cycle. In such environments, variability is guaranteed and uncontrollable; previous and current attempts to address software deployment in networked environments rely on the assumption that a single configuration of a software system is directly copied to a controlled set of consumer sites. Such assumptions do not apply to environments such as the Internet because of the variability that is inherently present. Some of the main areas of variability are platform, platform configuration, software configuration, and software revision.

Computing platform variability is obvious, but it still requires specific support. Computing platform configuration variability is not as obvious, but the notion that the same computing platform is equivalent to another as long as the operating system is the same is too simplistic. It is necessary to recognize that similar computing platforms may differ considerably due to their configurations. An often-neglected form of variability is the software system configuration variability that results from the implicit heterogeneity of usage in large network environments. It is not sufficient to assume that every consumer uses the same software system configuration as every other consumer. Therefore, it is necessary to describe the variation of the software system as well. Software configuration variation includes functional and non-functional variations within a software system, such as optional capabilities or performance options respectively.

The last form of variability, software revision, results from time-ordered changes to software systems by software producers. This variability is exacerbated by component-based software development technologies. Specifically, the push toward component-based software development results in the creation of software systems that are built using other software systems, referred to as a system of systems. Such software systems rely on independently developed and released software components for their functionality. This creates a nightmare of tangled revision dependencies among the systems, a byproduct of the component-based approach that is rarely discussed. The result is that the lifetime of a specific revision of a software component is significantly extended. Since the software producer has less control over which revisions of its software components are in the field, it cannot arbitrarily withdraw specific revisions without affecting other software systems that still require that revision.

Further, it is necessary to address two related issues in software deployment languages and schema, namely automation and policy. The degree of software deployment process automation that is possible or desirable as opposed to required or allowed is of central importance and requires a broad range of operating modes. Additionally, the deployment processes, whether automated or not, must parameterize policy decisions. A specific deployment process performs a specific task, whereas a policy indicates how a particular deployment process should perform its task. Flexibility in these areas is crucial for the wide-scale acceptance of any deployment language or schema.

5.3 Requirements

Software deployment is a collection of interrelated activities that address all of the issues of interfacing a deployed software system to the ongoing usage of the consumer as well as the ongoing development efforts of the producer. The software deployment activities include release, install, update, reconfigure, adapt, activate, deactivate, remove, and retire. These activities define the scope of the software deployment problem space; a space for which software deployment languages and schema must provide coverage.

The purpose of software deployment language and schema technologies is to provide knowledge that is rich and rigorous enough to support the automation of the software deployment life cycle. Given such descriptions, generic solutions to software deployment tasks are possible by combining software product knowledge with consumer site knowledge.

Descriptions of software systems and consumer sites require a semantic model of these entities. A common approach used by OSD, MIF, and AMS, is to model software systems and computing sites as nested collections of properties. These collections of properties may have internal structure, but at the base level they map to primitive data types (e.g., integer, string). This model is assumed and used throughout this dissertation.

5.3.1 Consumer Site Description Requirements

The purpose of the consumer site description is to provide context about the site into which a software system is situated; this is essential to fully describe a software system or component. For example, it is difficult to describe a software system's dependency on a property such as operating system, architecture, or dependent software subsystems if there is no accessible model of such information. As such, the consumer site description and the software system description are two halves to a whole, rather than two distinct entities.

At the very least, the consumer site description must record information about the state of the consumer site at any given time; such records include the following:

- Properties - attributes of the consumer site that may affect the outcome of the deployment operations (e.g., operating system, hardware architecture, and hardware configuration such as memory),
- Constraints - restrictions placed on the values of specific site properties either by deployed software systems or by administrators (e.g., a deployed software system may constrain the operating system configuration or the version of a shared resource), and
- Resources - interfaces and capabilities provided by the consumer site as well as the existence and availability of other deployed software systems at the site.

5.3.2 Software System Description Requirements

The other half of the deployment puzzle is represented by producer descriptions of software systems and components. These descriptions define an interface to the products and services provided by software producers. The goal for software system description is to provide enough information about a software system in order to deploy it in an automated fashion. The responsibility of describing a software system lies solely with the producer of the software system since they have the most knowledge about the system.

This thesis has identified five classes of information that are necessary to describe a software system for deployment; a description of each is presented in the following subsections.

5.3.2.1 Assert constraints

The correct operation of a software system is dependent upon values of properties at the site where the software is deployed; these types of assert constraints are not resolvable in the presence of a conflict. Some possible examples of assertions are the requirement that the operating system is Microsoft Windows 95 or that the screen resolution is greater than 800 pixels by 600 pixels.

Assert constraints are used for two different purposes: to select a properly configured software system for deployment and to maintain the proper operation of a deployed software system. Two common examples of the former type of assert constraints are hardware architecture and operating system; upon determining the operating system and architecture for a consumer site it is possible to select the artifacts to install.

The latter use of assert constraints is best described with an example. Imagine a particular deployed software system that requires version 1.0.2 of the Java Virtual Machine (JVM). This constraint was, by definition, true at the time of installation, but after installation it is possible to violate this constraint when the JVM is upgraded. The deployed software system no longer functions properly since its constraint on the JVM is no longer satisfied.

Assertions can alleviate this situation by not allowing changes that violate existing constraints. Assertions are generally discarded after they are used to select a properly configured software system. In order to guarantee operational correctness, it is necessary to maintain and manage the assertions of the consumer site, thus illustrating that a consumer site model is essential for deploying software systems.

Complications may arise from inherited assert constraints. For example, a deployed software system may constrain version 1.0.2 of the JVM. A new software system may constrain version 1.1.7 of the JVM. These constraints are in conflict and any attempt to deploy the second software system will fail because there is no way to resolve the constraint conflict. Assertions guarantee the failure of such a scenario, even if it is possible to install multiple versions of the JVM.

5.3.2.2 Dependency constraints

Dependency constraints are a type of constraint that provides a means to resolve a conflict if one arises. The more general assert constraint is still necessary, though, because not all constraints are solvable and even those that are solvable should not be solved in all cases. A common dependency constraint is present in a Web-based software system. This type of system depends on the existence of a Web browser in order for it to operate. It is possible to retrieve and install a Web browser if one does not exist; thus subsystem dependencies are one type of dependency constraint.

The line dividing assert from dependency constraints is subjective, though. Recall the previous example of two systems requiring different versions of the JVM. In this scenario the second software system could retrieve and deploy its required version of the JVM since the two versions of the JVM do not conflict with each other. This is a reasonable approach because the JVM consumes a relatively small amount of resources; therefore it makes sense to allow multiple copies. In other scenarios, though, multiple copies of a subsystem might not make sense because of incompatibilities or impracticalities of resource usage.

These scenarios emphasize the importance of having distinct notions of assert and dependency constraints. Assert constraints only assert that the constraint is true while dependency constraints try to find a resolution. If assert constraints were not treated separately from dependency constraints, each constraint conflict on a subsystem, for example, would request a new copy of the subsystem be deployed to meet the new constraint specification. It is clear that this is not a desirable situation; this approach would litter the consumer site with various revisions and configurations of the same subsystem.

The discussion of dependency constraints thus far has centered on the notion of a dependent subsystem and its version number. This is only one possible example of a dependency constraint. Within subsystem dependencies one might wish to constrain the subsystem based on some functional configuration, rather than the version number. Given a conflict it is possible to attempt to reconfigure the functional aspects of the dependent subsystem. As another example, a dependency may constrain the configuration of the consumer site operating system. In such a case it is possible to resolve a conflict by reconfiguring the operating system. The notion of a dependency constraint requires flexibility.

There are also different types of dependency constraints. By expanding beyond the notion of co-requisite dependencies, a class of temporal dependency constraints is found. Temporal dependency constraints exist at a particular time during a software system's deployment life cycle, but they are not integral pieces of the software system's time invariant definition. For example, a software system deployed at a particular site may have a prerequisite dependency on a specific tool during installation, such as an unarchiving tool. The unarchiving tool is only required during installation and therefore is not necessary for the proper operation of the software system after installation.

The notion of an abstract dependency constraint is also a common requirement. A software system that has documentation files in HTML format has an abstract dependency on an HTML viewer. From the software system's perspective it does not matter which HTML viewer is available, it is just concerned with the availability of the abstract capability of viewing HTML formatted documents. These types of abstract descriptions of software systems are very important and a complementary schema for describing abstract capabilities requires separate consideration. A simplistic approach to this issue is possible via MIME types for the Multipurpose Internet Mail Extensions protocol [18], but a more comprehensive approach is necessary.

5.3.2.3 Artifacts

When describing a software system it is necessary to describe the actual, physical components that make up the system. The physical components of a system are the collection of executables, libraries, data sets, and documentation that are used to compose the software system. Not all software systems will have all of the aforementioned types of component files, but they serve to illustrate the general classes of artifacts for a software system. The artifact description should include information such as location, description, and type of the artifact.

5.3.2.4 Configuration

There are two types of configuration information that describe a software system or component. The first, and simpler, of the two is the settings or configurable properties of a software system. A software system generally has many configurations. The various configurations may determine simple aesthetic issues or they may determine various levels of performance or functionality. It is therefore imperative to describe the variability of a software system so that other software systems that depend on it can examine its configuration for deployment purposes. By fully identifying and recording this configuration information it is then possible to uniquely identify the existence and exact state of the deployed software system.

The second type of configuration information is a higher level description of the system components. While a description of the software system artifacts is necessary to perform most of the basic software deployment tasks, a higher level description of the software system is required to perform some of the more abstract software deployment tasks. This high-level description of the software system's configuration is similar to an architectural description, but specifically it describes special relationships between specific software system components. For example, a client/server system must describe the relationship between the client program and the server program so that the activation activity of the software deployment life cycle can understand that it must activate the server before it can activate the client. In addition to relationships between components, the configuration description must also capture the interfaces and capabilities provided by those components. These interfaces may include management-related functionality as well as service-related functionality.

The relationship information provided in the configuration description is not used in isolation. The relationship information details *how* a software system is configured. This information then affects all other aspects of the software system description. For example, a software system may have many possible configurations. Choosing one configuration over another may result in changes to the specific set of constraints, artifacts, and activities for the given software system.

5.3.2.5 Activities

Despite the fact that the previous classes of information describe a large portion of a software system for deployment, it is still not possible to know all the specialized activities that are performed during various processes of the software deployment life cycle.

Full support of software deployment requires support for specialized activities that are required during the various software deployment processes. For an example, consider a software system that maintains an index on an evolving collection of Web pages. Each time any of the software deployment life cycle processes modify the collection of Web pages, by addition, update, or removal, it is necessary to re-index the Web page collection. It is not easy to infer this type of activity from the previous classes of information and therefore it requires separate description.

No matter how well thought out a deployment language or schema is, it is not possible to anticipate every specialized activity that a specific deployment process might require. Also, by supporting activity descriptions it is possible to understand the relationships between the activities and the deployment processes in order to reduce redundancy and increase correctness in the overall deployment solution.

Chapter 6

The Deployable Software Description Format

The Deployable Software Description (DSD) format described in this chapter of the thesis is an application of the Extensible Markup Language (XML) [7]. DSD is a vocabulary for describing software systems and their complex internal and external dependencies and relationships for heterogeneous software consumers. The end result enables the creation of software system descriptions that simplify or automate software deployment tasks. The general approach of DSD is described first, followed by a discussion of its XML document type definition (DTD), which describes its valid syntax.

6.1 Approach

The approach used by DSD to describe software systems for deployment is to create a standard language or schema. The OSD format, the Software MIF, and AMS as discussed in Section 2.3 also use this approach. These approaches model software systems and computing sites as nested collections of properties. Abstractly, it is possible to characterize DSD as nested collections of name-value pairs, but more specifically, DSD creates a standard, hierarchical schema representation to describe software systems for deployment.

The decision to use a language or schema approach requires the selection of a base unit of description. In the simplest case, a software system schema may describe a single software system configuration, i.e., a single revision and variant. In the most complex case, a software system schema may describe an entire software family, i.e., all revisions and variants. These two extremes are the endpoints of a spectrum in which various combinations and hybrids may occur. For example, a software system schema may alternatively describe all revisions of a single variant. The OSD approach describes selected variants of a single revision, where the MIF and AMS approaches describe a single variant and a single revision.

The approach taken in DSD is to describe an entire software family within a single schema description, i.e., all revisions and variants. There are a variety of reasons for choosing this approach. First, as discussed earlier, the lifetime of a single revision of a software system is seriously extended as a direct result of component-based software development. In a cooperative deployment environment, software producers are no longer able to simply retract old revisions of their software systems. It is necessary to maintain the availability of older software system revisions in order to provide backward compatibility with the dependent software systems of other software producers. Maintaining a range of revisions in a single description simplifies revision resolution and access.

The complete family description approach also provides more flexibility. With the complete family approach it is possible to describe a dependency on a specific configuration of a software system, rather than a specific instance of a software system. For example, a software system may depend on a certain capability of a software system, rather than a specific revision of a software system. By defining a configuration dependency on a capability it is possible to choose various revisions that meet the defined requirement, including past and future revisions. These types of configuration dependencies are difficult when the deployment language or schema partitions the description space by revision.

Additionally, the segmented approaches suffer in the area of reuse when compared to the complete family approach. A given software system may have significant portions of its description that are identical across many revisions and variants. A combined approach leverages existing descriptions as well as provides a repository to analyze and understand the relationships that exist between revisions and variants.

The complete family approach does create a scalability issue in terms of both mental effort and resource overhead. It is possible to lessen the mental effort associated with this approach by creating tools that create separate DSD specifications and then merge them into a family. Extending the DSD specification to include pointers or references to file artifact lists could lessen the resource overhead. It is important to point out that DSD is not restricted to the complete family approach. The definition of DSD is not rigid and any approach along the spectrum is possible. The adoption of the complete family approach is merely to provide direction for long-term benefits, not to dictate short-term usage.

DSD also makes an assumption that consumer site description information is accessible. The consumer site description must model, as discussed in Section 5.3.1, information such as computing platform, operating system, and installed software systems. DSD does not specify how the consumer site is modeled, but an accessible model is required in order for DSD to realize its fullest potential. Other approaches tend to include a limited number of standard consumer site properties or they merely ignore the issue all together.

Also implicit within DSD is an expression language. The expression language is not specified in this thesis, nor is the expression language explicitly specified by DSD at all. Any arbitrary expression language is appropriate as long as it is extended to allow expressions over the described software system properties and the consumer site properties. For example, if a software system has an “on-line help” property, then the expression language must provide access to the value of this property. Similarly, the expression language must provide access to the values of consumer site properties such as computing platform, operating system, and installed software systems. The expression language is largely used to place boolean guards strategically throughout a given DSD specification to create a mapping from the software system properties to the schema elements themselves. For example expressions and usage refer to Appendix B.

6.2 Document Type Definition Overview

The document type definition (DTD) presented here makes two simplifying decisions. First, the order of child elements in aggregated types is restricted to the order defined in the DTD. Second, the existence of child elements in aggregated types is restricted to the child elements defined in the DTD. These assumptions are made to simplify the XML DTD grammar. Only the appropriate portions of the DTD are presented here, the DTD is presented in its entirety in Appendix A. The creation of a DSD namespace using the XML namespace mechanism will be necessary to avoid naming conflicts.

6.2.1 Family

The `Family` type is the top-level or XML document schema element. As the type name suggests, `Family` describes a software family. The XML definition is:

```
<!ELEMENT Family (Id, ImportedProperties, Properties, Composition,
    Assertions, Dependencies, Artifacts, Interfaces, Notifications,
    Services, Activities)>
```

The main purpose of `Family` is to provide a container for the complete set of deployment information pertaining to a given software system family; the actual details of the description are provided within the elements of `Family`.

6.2.2 Id

The `Id` schema element is a container mostly provided for human-readable content. The XML definition for `Id` is:

```
<!ELEMENT Id (IdName, IdDescription, IdProducer, (IdLicense)?,
    IdSignature)>
```

The first three elements of `Id` are simple character string values that represent the unique name, description, and producer of the software family. The `IdLicense` element is an optional reference to a licensing document. The final element, `IdSignature`, is an MD5 content hashcode representing the current version of the family description. This code is updated whenever the family description is updated and is used to quickly determine whether a family description is equivalent to another instance of the family description.

6.2.3 ImportedProperties

The `ImportedProperties` schema element is a container for storing zero or more `ImportedProperty` elements. `ImportedProperty` elements represent external properties used in the family description. External properties refer to properties that are not defined in the family description itself, such as the operating system or architecture of a target deployment site. The XML definition for both `ImportedProperties` and `Imported-Property` are:

```
<!ELEMENT ImportedProperties (ImportedProperty)*>
<!ELEMENT ImportedProperty (PropertyName, PropertyType,
    PropertyDescription, PropertyValue)>
```

An `ImportedProperty` element is basically a name-value pair, where the "name" is the name of the imported property and the "value" is the value of the imported property at the time of usage; for completeness, property type and description are also specified. The imported property section is not completed during the description creation process, since imported property values are not known in advance, only the property names themselves are completed. The values of the imported properties are recorded at the time of deployment. It is important to keep track of these values explicitly, because in the advent of changes to these values, the original values are required to successfully reconfigure or undo previously performed deployment operations.

6.2.4 Properties and Property

The `Properties` schema element is a container for zero or more `Property` elements. A `Property` describes a specific internal property of a software family. The XML definitions for both `Properties` and `Property` are:

```
<!ELEMENT Properties (Property)*>
<!ELEMENT Property (PropertyName, PropertyType, PropertyDescription,
    PropertyDefaultValue, PropertyDefaultEnabled,
    PropertyDefaultDisabled, PropertyTopLevel, PropertyValues)>
```

The main pieces of a `Property` schema element are its name, type, description, and default value. The type of a property is a character string, a number, or a boolean. The default value associates a starting value for a software system property, but the final value is not part of the software system description, instead it is recorded separately when the software system is deployed. Many of the other elements of `Property` help when trying to process the properties in some automated fashion. Default values for enabling or disabling the specific property are specified in `PropertyDefaultEnabled` and `PropertyDefault-Disabled` respectively. The notion of a "top-level" property is introduced in order to create a pseudo-hierarchy of properties that is further expanded in the `Composition` section. The final element, `PropertyValues`, is a collection that is used to specify possible values for a given property.

It is important to understand the nature of a property with respect to the DSD and with respect to a software family itself. With respect to DSD, a property has no real meaning. A property is merely a configurable value that is used to organize pieces of the schema using schema guard expressions. With respect to the software family, a property may encode any meaning that is required. The most common example is that of a version number. By creating a version number it is possible to organize the family description in terms of a version numbering scheme. Another example might use a property to represent an optional performance variant.

Implicit in the above discussion is the fact that DSD places no meaning on a version number; this is a significant departure from many deployment and configuration management approaches. This decision was not based on any inherent deficiency in the version number notion, rather the intent was to make the approach more flexible for deploying software that does not necessarily lend itself to the version number notion, such as pure content delivery. A side effect is the realization that version numbers "encode" more than just the time-ordered sequence of a software system, and instead are also used to determine functionality and capabilities of a software system. Placing less importance on the version number and providing a means to describe capabilities as properties creates a more flexible and informative software description.

6.2.5 Composition and CompositionRule

The `Composition` schema element is a container for zero or more `CompositionRule` elements. A `CompositionRule` describes a specific relationship between two or more properties. The XML definitions for both `Composition` and `CompositionRule` are:

```
<!ELEMENT Composition (CompositionRule)*>
<!ELEMENT CompositionRule (RuleCondition, RuleControlProperty,
    RuleRelation, RuleProperties)>
```

The main pieces of a `CompositionRule` are the condition, relation, and property elements. The condition is a schema expression that returns a boolean result that signifies whether the rule applies; if the boolean result is “true” then the rule applies, if the boolean result is “false” then the rule does not apply. The relation element indicates the type of relationship that is described by this rule; the currently supported relationships are any-of, one-of, includes, and excludes. The first two relationships indicate that a rule requires the selection of any or one of the properties contained in `RuleProperties`, respectively. The second two relationships indicate that a rule explicitly includes or excludes the properties contained in `RuleProperties`. The `RuleProperties` element is a container of internal property names that participate in the relationship. For an example of how a composition rule is used, consider a rule that tests whether the operating system is UNIX and its relationship includes a “man page” property. In this scenario, the “man page” property is listed in the `RuleProperties` container of the composition rule. When the operating system is UNIX the composition ruled is applied. Since this is an “include” relationship, the “man page” property is included by setting its value to the default enabled value described in the `Property` schema element for each specific property.

The final element, `RuleControlProperty`, helps create a hierarchy of properties by specifying the name of an internal property that "controls" the specific relationship. A property controls the relationship if the result of the condition expression is based solely on the value of the property itself. The notion of a controlling property is useful, for example, when creating a user interface that attempts to display the relationships among properties. The adoption of a purely hierarchical representation of properties could have avoided some of the pseudo-hierarchical trappings in `Property` and here in `CompostionRule`, but would have resulted in less flexibility. In the current definition, properties may participate in arbitrary combinations of hierarchical or non-hierarchical relationships.

6.2.6 Assertions and Assertion

The `Assertions` schema element is a guarded container for zero or more `Assertions` and `Assertion` elements. An `Assertion` describes a specific constraint of the software family via a schema expression over internal and external properties. An assertion is intended to test a software family constraint that cannot or should not be resolved in the face of a conflict. The XML definitions for both `Assertions` and `Assertion` are:

```
<!ELEMENT Assertions (Guard, (Assertions | Assertion)*)>
<!ELEMENT Assertion (Guard, AssertionCondition,
    AssertionDescription)>
```

Both `Assertions` and `Assertion` are guarded with a `Guard` element. A guard is a schema expression that evaluates to either true or false. The expression is arbitrary and can reference both internal software family properties defined in the `Properties` section and can also reference external properties of the target deployment site. In both cases the guards are used to determine whether the specific schema element is applicable to the current, selected configuration of the software family. An empty guard is considered to imply applicability.

The AssertionCondition element of Assertion is also a schema expression; this expression is the actual assertion to test and verify. If the condition returns a false result, then the assertion fails and any deployment process in progress fails. Some very simple examples of an assertion are asserting that the operating system is “Win95” or that total memory is greater than “24MB.” In both of these instances, if the condition is not true, there is no way to resolve the conflict and thus failure is the only alternative. An assertion may also have a human-readable description attached to it.

6.2.7 Dependencies and Dependency

The Dependencies schema element is a guarded container for zero or more Dependencies and Dependency elements. A Dependency describes a specific constraint of the software family via a schema expression over internal properties and external properties. A dependency tests a software family constraint that is resolvable in the face of a conflict. The XML definitions for both Dependencies and Dependency are:

```
<!ELEMENT Dependencies (Guard, (Dependencies | Dependency)*)>
<!ELEMENT Dependency (Guard, DependencyCondition,
    DependencyDescription, DependencyResolution,
    DependencyConstraints)>
```

Both Dependencies and Dependency are guarded with a Guard element; guards are used to determine whether the specific schema element is applicable to the current, selected configuration of the software family. The DependencyCondition element is a schema expression that actually tests the dependency condition where a return result of true indicates that the dependency is satisfied and false indicates that it is not satisfied. The most common example of a dependency is a dependent subsystem where it is possible to install the subsystem if it is not present at the target deployment site. Another example is an operating system parameter where it is possible to reconfigure the parameter if it is not the desired value. In both of these instances, if the condition is not true, there is a specific means to attempt to resolve the conflict; the dependency only fails if the resolution attempt fails. The

DependencyResolution element is a placeholder to describe the resolution procedure. Constraints for the resolution process are placed in the DependencyConstraints element, which is simply a collection of bound properties. A dependency may also have a human-readable description attached to it.

6.2.8 Artifacts and Artifact

The Artifacts schema element is a guarded container for zero or more Artifacts and Artifact elements. An Artifact describes a specific software system artifact, represented as a file, of the software family. Artifacts are the main building block of a software family and any specific software system configuration. The XML definitions for both Artifacts and Artifact are:

```
<!ELEMENT Artifacts (Guard, (Artifacts | Artifact)*)>
<!ELEMENT Artifact (Guard, ArtifactSignature, ArtifactType,
    ArtifactSourceName, ArtifactSource, ArtifactDestinationName,
    ArtifactDestination, ArtifactEntryPoint, ArtifactMutable,
    ArtifactPermission)>
```

Both Artifacts and Artifact are guarded with a Guard element; guards are used to determine whether the specific schema element is applicable to the current, selected configuration of the software family. The ArtifactSignature element is a content signature, such as MD5, that uniquely identifies the described artifact. ArtifactType is a simple type mechanism for the artifacts; certain artifact types may imply specialized handling techniques.

There are parallel elements in the Artifact specification for dealing with the name, source, and destination of a particular artifact. The ArtifactSourceName element specifies the name of the artifact as it appears in the source for the specified artifact. The ArtifactSource element may reference a directory or an archive that contains the artifact. The ArtifactSource is specified as a direct path or via a URL. The ArtifactDestinationName specifies the name that the artifact must have in its final destination; the destination name and the source name may differ. The `ArtifactDestination` element specifies the destination directory for the artifact that is relative to the base directory where the software system is installed; the destination is relative since the installation location varies for each deployment.

The `ArtifactEntryPoint` element is a boolean value that indicates whether a particular artifact is an entry point into the software system, such as an executable. If an artifact is marked as an entry point, this information is used to create access points in the user environment, such as placing an icon on the user's desktop. `ArtifactMutable` is a boolean value that specifies whether the artifact may change once it is deployed. If an artifact is mutable then subsequent deployment processes have to take mutated files into account when they are performing their operations. The last element, `ArtifactPermission`, is a UNIX-style permission mask that specifies the default permission on the file. Operating systems that do not support file permission may ignore the permission element; an empty permission attribute is assumed to grant full permission.

6.2.9 Interfaces and Interface

The `Interfaces` schema element is a guarded container for zero or more `Interfaces` and `Interface` elements. An `Interface` describes a specific interface provided by the software system. The `Interface` element is not intended to depict low-level interfaces implemented by the software system, rather it is intended to depict high-level interfaces, such as deployment operations or administration tasks. The XML definitions for both `Interfaces` and `Interface` are:

```
<!ELEMENT Interfaces (Guard, (Interfaces | Interface)*)>
<!ELEMENT Interface (Guard, InterfaceName, InterfaceDescription)>
```

Both `Interfaces` and `Interface` are guarded with a `Guard` element; guards are used to determine whether the specific schema element is applicable to the current, selected configuration of the software family. The `Interface` element has an `Interface-`

Name element that specifies the name of the interface and an InterfaceDescription element that specifies a description of the interface. It is important to recognize that the methods for invoking or activating an interface are not specified by the DSD, rather that an Interface is merely a means to describe that such an interface exists.

6.2.10 Notifications and Notification

The Notifications schema element is a guarded container for zero or more Notifications and Notification elements. A Notification describes a specific notification that is generated by the software system. The Notification element is not intended to depict all low-level events that are generated by the software system, rather it is intended to depict high-level notifications, such as those generated by deployment operations or for administration purposes. The XML definitions for both Notifications and Notification are:

```
<!ELEMENT Notifications (Guard, (Notifications | Notification)*)>
<!ELEMENT Notification (Guard, NotificationName,
    NotificationDescription)>
```

Both Notifications and Notification are guarded with a Guard element; guards are used to determine whether the specific schema element is applicable to the current, selected configuration of the software family. The Notification element has a NotificationName element that specifies the name of the notification and a NotificationDescription element that specifies a description of the notification. It is important to recognize that the methods for sending or receiving a notification are not specified by the DSD, rather that a Notification is merely a means to describe that such a notification is generated.

6.2.11 Services and Service

The Services schema element is a container for zero or more Services and Service elements. A Service describes a specific service provided by the software system. Service is intended to depict high-level services that a software system exports to its networked environment, unlike interfaces that are local to their respective site. An example of such a service may include an ODBC data source or an HTTP server. The XML definitions for both Service and Services are:

```
<!ELEMENT Services (Guard, (Services | Service)*)>
<!ELEMENT Service (Guard, ServiceName, ServiceDescription)>
```

Both Services and Service elements are guarded with a Guard element; guards are used to determine whether the specific schema element is applicable to the current, selected configuration of the software family. The Service element has a ServiceName element that specifies the name of the service and a ServiceDescription element that specifies a description of the service. It is important to recognize that any form of service utilization is not specified by the DSD, rather that a Service is merely a means to describe that such a service is provided.

6.2.12 Activities and Activity

The Activities schema element is a container for zero or more Activities and Activity elements. An Activity describes a specialized deployment activity that is required by the software system. An example of such an activity is re-indexing a collection of Web pages after an update has occurred. The XML definitions for both Activities and Activity are:

```
<!ELEMENT Activities (Guard, (Activities | Activity)*)>
<!ELEMENT Activity (Guard, ActivityAction, ActivityWhen,
    ActivityDescription)>
```

Both Activities and Activity elements are guarded with a Guard element; guards are used to determine whether the specific schema element is applicable to the current, selected configuration of the software family. The Activity element has an Activity-Action element that is a placeholder to describe the action to perform. The Activity-When element describes when the activity should occur; the contents are not specified by

DSD but a reasonable example is “after update.” The `ActivityDescription` element specifies a description of the activity. It is intended that standard deployment processes will perform the corresponding action of the `Activity` schema elements when appropriate.

6.3 Creating a Deployable Software Description Specification

In order to use the DSD format, it is necessary to have a certain level of familiarity with its structure and type system. The DSD format is not strict, and therefore it allows collections to contain any other available type. The purpose of this is to allow flexibility and extensibility. To create standard, generic deployment processes, though, it is necessary to place some binding rules on the structure of a DSD specification. For example, the generic deployment processes in the prototype Software Dock framework assume that all artifacts are described in the artifact collection of the family; any artifacts described elsewhere are simply ignored. Therefore, the general rule of thumb when creating a DSD specification for the Software Dock’s generic deployment processes is that the entire specification should follow the internal structure of the `Family` type. Any deviation from this structure is simply ignored by the generic deployment processes.

The first step to creating a DSD specification is to determine the proper set of properties to use to organize the software system. There is no simple rule to follow to determine which properties are most appropriate for organizing a software system. Typically, the Microsoft Windows 95 platform tends to organize software systems with respect to capabilities or features. It is also worthwhile to use properties to organize a software system with respect to architectural components or to strict revision numbers. DSD does not require a specific property approach for organizing the software system; the properties of the software system depend on the software system itself.

Once the organizational properties for the software system are defined, it is necessary to describe the relationships among the properties. Composition rules are used to describe the relationships among properties. These relationships define the basic behavior that is allowed when manipulating the software system configuration. For example, a property can include or exclude a secondary set of properties. Using the composition rules, it is possible to create generic algorithms that manipulate a software system in order to select valid configurations.

With the composition rules defined it is necessary to define the remaining elements of the DSD specification with respect to the software system properties; the guard statements attached to most DSD schema elements serve this purpose. The schema elements that have guards associated with them use this guard to express when they are applicable to a given configuration. For example, if a file artifact that implements online help documentation is only applicable when the “online help” property is true, then a guard is attached to that file artifact to specify this condition. All remaining aspects of the DSD specification are mapped onto the properties using this technique. This technique is not limited to software system properties, though, and is also used to map schema elements to specific consumer site properties, such as architecture and operating system. Since most schema elements have associated collection elements that can also have guard expressions, it is not necessary to attach a guard to each individual schema element; common guard expressions are attached to the parent collection instead. An empty guard expression implies that its associated schema element is always applicable.

Chapter 7

Software Dock Deployment Process Definitions

In the prototype Software Dock framework, agents implement the software deployment processes. In general, the other components in the Software Dock architecture are passive elements, such as data and interfaces. Agents, on the other hand, are active since they perform the functionality of the software deployment life cycle processes. The Software Dock framework enables the creation of a collection of generic agents that perform many of the standard software deployment processes, such as install, update, adapt, reconfigure, and remove. These generic agents, although useful in many cases, may not be sufficient for every case and therefore are also useful as base classes for the creation of other, more specialized deployment agents.

All agents perform their deployment processes by encoding some functionality that is then parameterized by the information provided in the DSD specifications and the consumer site descriptions. In this fashion, a single agent definition is used for any software system described using DSD and at any consumer site that has a field dock. The remainder of this section describes the generic deployment process algorithm that all current deployment agents perform and then describes each specific deployment process in more detail.

7.1 Generic Software Deployment Process Definition

As described in Chapter 6, 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 algorithm.

Most software deployment processes can be characterized as the transformation of one software system configuration to another based on the set of property values for a given software system configuration. A valid set of software system property values represents a particular valid configuration of a software system. Given a new set of valid property values, a deployment process simply transforms its current configuration to the new configuration by performing differential processing over the applicable schema elements of the DSD specification. The applicable schema elements for a software release are computable via the guard conditions that are dispersed throughout the DSD specification. Differential processing of the applicable schema elements creates a new software system configuration that corresponds to the desired software configuration. For a simple example of differential processing, consider if the version of a software system is changed from “1.0” to “1.1.” All of the artifacts associated with version “1.0” are removed, the artifacts associated with version “1.1” are added, and any common artifacts are left untouched.

The following pseudo-code represents a very high-level description of the generic deployment process algorithm. Error handling is largely ignored in the pseudo-code, but a transactional notion is introduced in order to indicate the critical boundary for error handling. The install, update, reconfigure, adapt, and remove software deployment processes all follow this general, abstract algorithm. For a detailed exploration of this algorithm, refer to Section 10.5.

```
procedure deploymentProcess(Family family) {
    DSDSpecification dsd = retrieveFamilyInstance(family);
    PropertyValue[] oldConfig = retrieveCurrentConfiguration();
    PropertyValue[] newConfig = selectNewConfiguration(dsd);
    Assertion[] newAssertions = calculateAssertsions(dsd, newConfig);
    Dependency[] oldDependencies = calculateDependencies(dsd, oldConfig);
    Dependency[] newDependencies = calculateDependencies(dsd, newConfig);
    Dependency[] removeDep = calculateRemovedDependencies(
        dsd, oldDependencies, newDependencies);
    Dependency[] addDeps = calculateAddedDependencies(
        dsd, oldDependencies, newDependencies);
    Artifact[] oldArtifacts = calculateArtifacts(dsd, oldConfig);
    Artifact[] newArtifacts = calculateArtifacts(dsd, newConfig);
    Artifact[] removeArtifacts = calculateRemovedDependencies(
        dsd, oldArtifacts, newArtifacts);
    Artifact[] addArtifacts = calculateAddedDependencies(
        dsd, oldArtifacts, newArtifacts);
    beginTransaction();

    updateFamilyInstance(newConfig);
    testAssestions(newAssertions);
    removeDependencies(removeDep);
    resolveDependencies(addDep);
    removeArtifacts(removeArtifacts);
    retrieveArtfacts(addArtifacts);

    commitTransaction();
}
```

7.2 Specific Software Deployment Process Definitions

The software deployment processes vary from each other in small, but important ways. Each specific deployment process is described below. There is an interesting, implicit issue with respect to all of the deployment process implementations described below. All of the agents manipulate the DSD specification of a given software release in isolation of the software system itself. This means that an agent needs only the specification of a software release to perform a large portion of its tasks. As a result, an agent is much more efficient, especially in the area of transfer time, since by manipulating the schema description first, the agent only requests exactly what it needs to finish its task. This is possible since the release dock works in cooperation with the agents to perform the deployment processes.

7.2.1 Install Process

The install agent deploys a new configuration of a software release to a consumer site. The install agent differs from the other software deployment process agents since it is not associated with an existing software release configuration. The install agent performs its task by first retrieving the current DSD specification for the software family for which it is responsible. The install agent queries the local field dock and the user to determine the configuration of the software release to install. 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. Once the install process is complete, the install agent is no longer needed and therefore it removes itself. Multiple install requests are always handled by separate install agents and therefore always install another configuration of the associated software release. If a software release is unable to have multiple installations at a site, it is necessary to add an assertion to the DSD specification that tests for this condition. Currently the install process is always invoked either directly or indirectly by a specific user request to install a software release; therefore the install process is always "pull" oriented.

7.2.2 Update Process

The update agent deploys a new, previously unavailable configuration of a previously deployed software release, thus eliminating the previously deployed configuration. The newly available software release configuration is provided in an updated DSD specification for the software release. The update agent must retrieve the new DSD specification from its release dock in order to perform the update. The update agent must account for the existing deployed software release by performing differential processing on the applicable schema elements for the existing and updated software release configurations. Differential processing requires the undoing of schema elements corresponding to the prior configuration and performing the associated actions of the schema elements for the updated configuration. Any schema elements that are shared among configurations are left untouched. A specific update agent always handles the update process for a specific deployed software release. The update process is either 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 is discovered or specifically selected by the user. An update is not always the result of a change to the currently selected configuration; a content-only update is also possible. In such a scenario, the update does not change the selected configuration of the software system, only the content of the current configuration. This is typical in many software systems that use a "channel" or content delivery model. Finally, an update may not actually update the deployed software release at all; an update may simply provide a new, more accurate DSD specification for the deployed software release.

7.2.3 Reconfigure Process

The reconfigure agent changes the current configuration of a deployed software release, thus eliminating the previously deployed configuration. The reconfigure agent differs from the update agent because it does not retrieve a new DSD specification from its release dock even if one exists; therefore the reconfiguration agent cannot perform an update. The reconfigure agent only manipulates the existing DSD specification of the deployed software release with which it is associated. The reconfigure agent determines the new software release configuration much like the install agent. Once a new configuration is chosen from the existing DSD specification, the reconfigure agent performs differential processing on the applicable schema elements much like the update agent. A specific reconfigure agent always handles the reconfigure process for a specific deployed software release. Currently the reconfigure agent operates in "pull" mode.

7.2.4 Adapt Process

The adapt agent maintains the consistency of a currently deployed software release configuration in the context of the consumer site. The adapt agent does not change the software release configuration at all, it enforces it. When invoked, the adapt agent uses the existing DSD specification for its associated software release to verify that the deployed software release matches its description. It does this by determining the applicable schema elements for the deployed software release configuration and then testing them to make sure that they are still valid. If any discrepancies are discovered, the adapt agent simply performs the default processing on the invalid schema elements in order to correct the problem. A specific adapt agent always handles the adapt process for a specific deployed software release. Currently the adapt agent operates in "pull" mode. The adapt agent is easily extended to operate in "push" mode where consumer-side events, such as file deletions, automatically instigate the adapt process.

7.2.5 Remove Process

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 must fail. A specific remove agent always handles the remove process for a specific deployed software release. One remove request may cause multiple remove requests to other remove agents in the case of dependent software releases. Currently the remove agent operates in “pull” mode.

Chapter 8

Out of Scope

The scope of software deployment is very large and very broad. As indicated in previous sections, software deployment cuts across and combines numerous research and technological areas. In order to provide a reasonable scope for this thesis, certain topics were deemed out of scope. These areas, while not summarily dismissed, were not given the focus necessary to consider them when evaluating the contribution of this dissertation. These topics are described in the following subsections along with a discussion of how the Software Dock might address them.

8.1 Security and Privacy

Mobile agents cause a large security concern because they may 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 [31]. The interface provided by the local field dock is the only interface that an agent has to perform its tasks. To extend the interface provided to agents, the field dock uses a capability approach. A capability approach allows the field dock to offer agents access to certain restricted operations, such as controlled access to the local file system. In this capability-based scenario, an agent "docks" at the field dock and requests access to a protected resource. At the field dock's discretion, an object that provides an interface to the protected resource is granted to the agent, thus the agent is given a capability to the protected resource. Currently, the JVM does not support a true capability approach, but this functionality is expected in version 2 of the Java Development Kit. Regardless, all current agents are implemented as though this security approach was in effect; thus there is a relatively simple transition when support for the capability-based security approach is released.

An extension to the capability approach described above can enable the field dock to offer further levels of restricted operations to agents. Using security signatures to indicate trusted sources, it is possible to create a mechanism that offers more security sensitive capabilities to agents from trusted sources.

Trusted agents provide a mechanism to extend the local field dock in arbitrary ways without the need to evolve the actual interface that the field dock exports. This is due to the event-based registry in the field dock itself; every operation on registry data generates an event. Agents can subscribe to these events and perform subsequent actions upon receipt. Any agent may insert nodes into the field dock registry and then semantically map registry events to the specific operations that the agent is able to perform. This method creates a form of remote procedure call, where a collection node in the field dock registry is similar to an interface definition and inserting a node into that collection node is similar to making a procedure call. The removal of the inserted node from the collection node indicates the procedure return. The registry interface provides a variety of mechanisms for passing additional information through its event interface, thus creating a means to pass arbitrary parameters into these simulated procedure calls. This mechanism is not limited to trusted agents, even distrusted agents may use this technique, but the operations that they perform are limited to their level of security.

Privacy issues are much more complicated than security issues. Privacy is concerned with limiting who can actually see the information. In the case of software consumers, they must expose the configuration of their sites to incoming agents. It is quite likely that the consumer does not want configuration knowledge, such as which software systems they have installed, made publicly available. Privacy is not a concern for the standard agents since their implementations are known and trusted. Privacy is very complicated in the case of customized agents though. Since custom agents may legitimately require access to sensitive consumer information to perform their task, they cannot be unilaterally forbidden access. It is possible to limit the types of backward communication that an agent can perform in an effort to stop information from flowing out of the consumer site, but clearly some form of backward communication is necessary and any form of communication can potentially be used to perform covert actions. Other approaches might involve adding additional DSD parsers that disallow an agent to ask questions that are not specifically related to its DSD specification, but bogus DSD specifications circumvent this approach. A more brute force approach could require that custom agents be shipped as platform-independent source code, such as Java, where the code is automatically analyzed with respect to privacy issues and then compiled locally. It is clear that there is no simple solution to issues of privacy and that much research is necessary before it can be dealt with in a sufficient manner.

8.2 Electronic Commerce

The Software Dock framework is open to electronic commerce considerations, which deal with performing financial transactions over networks, although they were not investigated. Additional agents at the release and field docks can extend the Software Dock framework to support electronic commerce related activities. For example, the installation agent could request a license before it installs or to ask for credit card information to allow the consumer to purchase a license. 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; a variety of approaches are possible.

8.3 Error Recovery

The environment in which the Software Dock operates is highly unreliable, considering the level of distribution and collaboration that is required. Due to this unreliability, error conditions are certainly going to occur. The current level of error recovery in the Software Dock prototype is minimal. In most cases if a process fails the agent merely attempts to undo the operations that it has performed. Error recovery was not applied in a rigorous manner in the prototype therefore catastrophic failures may leave a deployed software system in an inconsistent state. Fortunately, the adapt process can verify the state of a deployed software system and can then try to repair any errors that it finds; this is useful in the case of a failure that was due to an intermittent network failure. If the failure is due to an improper configuration of the deployed software system, the reconfigure agent can select a different configuration that may work properly.

Adding true transactional capabilities to the Software Dock registries and the file system interface could significantly improve error handling. Transactional capabilities make it possible to reduce the instances where a deployed software system is left in an inconsistent state because they ensure that operations are properly undone when an error occurs. A sophisticated distributed transaction mechanism is necessary to ensure that all associated actions are undone, because the actions of a specific agent may cause side effects through other agents that have subscribed to events and perform actions in response to them. As a result, any agent that performs concomitant actions in response to an event would have to join the transaction associated with the event and then behave accordingly in order to maintain consistency.

8.4 Deployment Process Interactivity

The issue of deployment process interactivity was not obvious initially and was discovered through experience with the Software Dock prototype. In some scenarios software deployment process interactivity is related to error recovery, but this is not always true and therefore is an issue in itself. The real issue behind software deployment process interactivity deals with the fact that a proper mechanism to get additional information from a human being, when necessary, is not available to an agent in some scenarios. In the simplest case, an agent is directly manipulated by a human being and therefore can simply ask the user for information when needed. This solution is not available in cases where the agent is operating autonomously and there is no associated user driving the process, such as with a “push” update.

This issue arises in the more complex administration scenarios as well, where an administrator is remotely performing operations on groups of sites. As agents traverse to remote sites to perform their tasks, their communication path back to the administrator becomes more complex. This is particularly true in error situations where the administrator might need to direct the agent to find a proper solution to the failure. The simple solution to this issue is to simply have an agent fail when it needs additional information and it does not have a means to ask for it. This does not provide a very robust deployment framework, though, so some effort to handle this situation in a more elegant fashion is necessary.

Chapter 9

Evaluation

This section defines a framework for characterizing and evaluating software deployment solutions. This framework is used to evaluate the prototype Software Dock. Related software deployment solutions are also critically evaluated with respect to these and other requirements. Finally, three usage experiments are described that illustrate the effectiveness of the Software Dock approach.

9.1 Deployment Solution Characterization and Capabilities

In order to evaluate a specific software deployment solution, it is necessary to characterize the requirements and capabilities that a software deployment solution should possess. These requirements and characteristics do not imply a specific architecture or approach; rather, they illustrate what is necessary to create a complete, unified software deployment solution. These characteristics, however, do define a vision for the future of software deployment.

A complete, unified software deployment framework creates an end-to-end solution for deploying software from the software producer to the software consumer. It redefines software development to include the software deployment life cycle as an integral part of the development process. The combination of the development life cycle and the deployment life cycle creates the complete software life cycle.

Software deployment is a shared responsibility between the software producer and consumer. Until recently software producers neglected this responsibility, despite the fact that they possess the required knowledge about their software systems to perform it. Much like a manufacturer is responsible for the ongoing proper functioning and repair of the items it produces, it is also the responsibility of software producers to ensure the proper functioning and repair of the software systems they produce. To enable this, an environment where both the software producer and the consumer can cooperate for the common goal of software deployment is necessary. This environment must facilitate open communication, negotiation, and cooperation among software producers and consumers. The following subsections describe requirements and capabilities necessary to achieve this goal.

9.1.1 Abstractions

One approach to characterizing a software deployment solution is in terms of the abstractions that it provides to unify the problem space. The worst case for a software deployment solution is where L x M x N different deployment activity specifications are required. In this case, L is the number of possible target sites, M is the number of product variants, and N is the number of different activities covering a complete deployment life cycle. Informally, this case is similar to having a deployment system consisting of a separate "Makefile" for doing some activity, such as installation, for every product into every known site. Though this is not a common situation it illustrates how to evaluate a software deployment solution with respect to how much it reduces the cardinality of L x M x N.

Given such an analysis, it is clear that the three main components that benefit from abstractions are: consumer, software system, and process. These abstractions define a means for evaluating the level of support provided by a particular deployment system.

9.1.1.1 Consumer Abstraction

The consumer abstraction creates a single, common interface to interact with the spectrum of possible consumer sites. If a given software system is deployable on a variety of target sites, the consumer abstraction removes the need for the producer to know how to communicate with each specific target. Instead, the producer will interact with a Microsoft

Windows 95-based system in the same manner as a UNIX-based system using this common abstraction.

Mechanisms such as GNU's Autoconf and the Microsoft Registry are two examples of consumer abstraction. Autoconf is used to produce a single program, "configure", which dynamically computes a consumer abstraction. The Registry, in contrast, is a passive repository containing the consumer abstraction. In either case, the deployment process is simplified since a producer can construct installation scripts that are parameterized by common information available from the abstraction. It is important to note, though, that these two examples specifically target different operating system platforms and, as such, do not provide a single, common interface.

The consumer abstraction largely provides query or discovery mechanisms for a site. These particular mechanisms provide access to information about the configuration and resources at a site. The consumer abstraction is not limited to obtaining descriptive information and may indeed abstract processes that have common functionality across all targets, such as file system access.

9.1.1.2 Software System Abstraction

The software system abstraction creates a single, common means to interact with and reason about software systems. While the consumer abstraction is used to obtain and describe the site where a software system is deployed, the software system abstraction is used to describe a complementary set of information about the software system itself.

In the simplest case where a software system is a single executable, then describing it completely is a simple process. The software system description is nothing more than an inventory of files, documentation pointers, contact pointers, and platform requirements.

Complex software systems pose the biggest challenge and have the greatest need for the software system abstraction. A complex system may have multiple, distributed components where subsystem dependencies between components are explicitly required. This type of dependency and constraint information has a direct impact on how deployment processes are performed. It is also possible that a software system has variant configurations that are dependent upon the resources available at the target site; it is necessary to capture all of this information in the software system abstraction.

The software system abstraction is not limited to executable software systems. Software systems solely based on data are also covered by the abstraction. A good example of this type of system is any document-based or content delivery-based system.

9.1.1.3 Process Abstraction

The process abstraction creates a single, common means to interact with and reason about deployment processes and activities. This layer of abstraction deals with the distinction between a process and policy. A deployment process, such as installation or update, is a set of steps that is followed to perform the specific deployment process. The steps that are performed characterize a process, whereas how the steps are performed characterize a policy.

For example, the update process has a distinct set of steps that it must take to actually perform an update. These steps include examining the target site's configuration, retrieving the necessary updated artifacts, and properly modifying the target site. Every update process will perform something similar to these basic steps. A policy is used to determine how these steps are carried out. A policy might indicate that updates should only occur during non-business hours or that someone in the system administration department must first approve updates. Given this distinction, it is clear that processes are parameterized by policy decisions.

9.1.2 Process Coverage

Another characteristic that is important for evaluating the level of support provided by a particular software deployment solution is process coverage. As defined in Chapter 3, the software deployment life cycle is actually a collection of interrelated processes. The process coverage provided by a particular software deployment solution is orthogonal to the abstractions that the solution provides. For example, Castanet provides consumer and producer abstractions, but they are overly simplistic and result in process coverage limited to the direct support of simple installation and the differential update of content. Access to the underlying computational engine (i.e., the computer) is used to enable support for other software deployment processes. This "Turing machine" approach is common, and sometimes necessary, but it clearly does not provide any implicit process coverage.

For evaluation purposes, the broader the process coverage provided by a software deployment system the better the solution. Despite that process coverage is orthogonal to the abstractions described above, it is expected that a well defined set of abstractions is necessary to provide generic, broad process coverage.

9.1.3 Coordination

Support for deployment process coordination is important. The growing abundance of software systems that implement distributed system models, such as client/server software, requires software deployment solutions to have a “global” view of the activities they perform. These distributed software systems contribute to the growing complexity of software deployment because their architectures have inherently complex and unreliable relationships. In addition to distributed systems, coordination is also required for deployment processes in an organizational setting. Consumer organizations that are comprised of many computing sites require coordinated deployment activities across groups of machines, such as updating an accounting system for the accounting department. To perform a task, such as this, where the desired deployment process is actually a set of individual deployment processes, it is necessary to support a global view where coordination occurs.

9.1.4 General Requirements

Capability-oriented requirements that fall outside of the general, abstract characteristics described above are described in this section. These capabilities are necessary to create an effective software deployment solution. This discussion is not exhaustive, but it discusses the most important capability requirements.

Internet scalability: The explosive popularity of the Internet has created a new, lucrative environment for software deployment. It is imperative that proposed software deployment solutions explicitly support large numbers of producers and consumers, distributed over large geographical distances. The scale of the Internet is many magnitudes larger than local-area networks and organizational intranets and therefore requires that special attention is paid to its requirements.

Raise Abstraction Levels for Software Deployment: Many of the current software deployment systems perform a specific process or subset of the software deployment life cycle. Unfortunately, these systems are usually limited to the specific task or set of tasks for which they were designed. Support for other deployment tasks is usually in the form of access to the underlying Turing machine (i.e., the underlying computer); this is not a very useful level of support. A software deployment solution must provide a framework that raises the level of abstraction for performing deployment-related activities. Raising the level of abstraction enables efficient and timely solutions to deployment activities without requiring the reinvention of tedious, error-prone infrastructure.

Provide Unified Access to Procedural Resources: Most software deployment activities require more than just declarative information. Many deployment activities perform system specific processing at the consumer site. Therefore, a software deployment system is not complete unless it attempts to provide controlled access to procedural resources of a consumer site as well as its declarative information. It is important for the model of a deployment solution to include access to procedural resources not only because it is necessary for many deployment tasks, but also because it can provide a measure of security for the underlying consumer site. Since access to procedural resources is provided by the deployment solution, it is possible to place restrictions on which activities are performed. This is not possible in approaches that merely grant access to the underlying Turing machine. Additionally, unified access to procedural resources enables arbitrary extension to the deployment solution in order to integrate unforeseen or specialized capabilities.

Explicit Bi-directional, Semi-continuous Communication: It is important to exploit the bi-directional, semi-continuous communication between software producers and consumers that exists via the Internet. The connectivity afforded by the Internet enables producers and consumers to participate in a symbiotic relationship where information flows between the two participants. This communication model enables support for the complete software deployment life cycle because software producers and consumers can cooperate. As a direct result, the consumer's quality of service is improved through mechanisms such as the notification or automatic installation of bug fixes and updates. For the producer, it is possible to receive direct feedback from consumers, such as when a software fault occurs, and to make an appropriate response, such as issuing a bug fix.

Autonomy: Since organizational boundaries and cultures are very distinct, it is very important that a software deployment solution creates an environment in which those differences can coexist. In particular these differences exist from consumer to consumer and from producer to producer. Consumers should have control over how their site is accessed and how deployment processes are performed. Producers should have control over the definition of their deployment processes without regard to how the organizations of dependent software systems perform their deployment processes.

Platform Independence: Many systems address particular aspects of software deployment, some systems even come close to addressing the entire software deployment life cycle, but few systems do so in a platform independent manner. Platform independence is a necessary prerequisite largely inspired by the global marketplace created by the Internet. In an effort to take advantage of such a market it is necessary to not make limiting assumptions about who or what the consumer may be. A consumer side abstraction is directly related to platform independence, though the two are not equivalent.

9.2 Mapping the Software Dock to Requirements

In order to evaluate the Software Dock deployment framework, it is important to understand how it maps to the specific requirements defined above. The first requirements to examine are the abstractions that the Software Dock provides. The Software Dock provides a consumer abstraction through the field dock and the field dock standard registry. The field dock itself provides access to certain specific abstractions, such as file system access, but the standard field dock registry is also available via field dock interfaces and it provides access to consumer site information. Additionally, the event procedure call mechanisms of the registry create a common procedural abstraction. As a result, it is possible to create software deployment processes that can interact with any consumer site.

The DSD format specifically addresses the software system abstraction. The DSD format defines the notion of software release properties, property composition, assertions, dependencies, artifacts, interfaces, notifications, and activities for software deployment. The Software Dock framework can deploy any software release that is described using the DSD format through standard, generic deployment processes. The reason that this is possible is because the DSD format provides a single, common abstraction where a deployment process needs only to understand how to manipulate the DSD specification, rather than the software release itself.

The standardized registries of the release and field docks and the DSD specifications provide a mechanism for process abstraction. Additional information, that represents process parameters, is provided in the DSD specification and the field dock registry. Generic processes can interpret these process parameters accordingly. Some simple examples include process parameters that indicate whether reconfiguring a software release is allowed or whether software release updates should occur at a particular time rather than immediately.

In the Software Dock, agents realize the process abstraction by defining a generic deployment process that is parameterized by the registry and DSD information. These agents provide a base class from which other, more specialized, agents are derived.

The DSD format, the field dock registry, and the standard deployment agents are the mechanisms used by the Software Dock to meet the process coverage requirements. DSD specifications are repositories of deployment process information. All of the information necessary to deploy a specific software release is included in its DSD specification. The standard DSD format provides description capabilities for a large number of systems, but it is an evolving definition. The DSD was designed for easy augmentation in order to refine specific processes or to perform new processes as necessary. The field dock registry provides the context into which a DSD specification is realized. The standard deployment agents combine DSD specifications and field dock registry information with generic algorithms to perform the variety of software deployment life cycle processes.

The Software Dock framework provides many lower level mechanisms to support deployment process coordination. The registries act as common repositories to store information necessary for coordinating groups of processes. Besides the data storage mechanism provided by the registries, the event mechanism provided by the registries also enables coordination. Currently the standard deployment agents use events to enable coordination among the deployment processes. Coordination through events is further extended by the event procedure call mechanism; this mechanism allows agents to create arbitrary interfaces in the registry that other knowledgeable agents may use at their discretion. The choice of agents as an enabling technology also simplifies some coordination activities since they are mobile and may move from site to site to perform a specific task. At a higher level, the interdock component of the Software Dock framework supports coordination. The interdock is a means to logically group related field docks. Independent field docks then use their interdocks as a means to share information and resources with one another through declarative information in the interdock registry. For example, in a client/server system, the deployment process for the server system places a service flag in the interdock's registry. Whenever a client system is installed, its deployment process searches the interdock registry for the service flag to determine whether the required service is available.

Internet scalability is an important requirement that is specifically considered in the Software Dock framework. The avenue in which the Software Dock pursues scalability is through the use of hierarchical naming and the event system in its field dock registries. A wide-area event publish-and-subscribe system, which is part of a related research effort [9], is the preferred means to extend local events to the Internet. Similar publish-and-subscribe approaches are used in other fields, such as financial markets, and are therefore believed viable for the Software Dock. The Software Dock employs a federated database approach through its field dock registries, thus allowing the number of software consumers to scale indefinitely; the same is also true of the release dock registries for software producers. It is problematic, though, that all deployment agents refer back to their originating release dock. For widely deployed software systems, literally thousands of agents acting on behalf of thousands of consumers might all descend upon a single release dock in response to an update event. It is necessary to introduce mechanisms to cope with these scenarios, such as Internet caching, or to re-implement the generic agents to have more courteous algorithms, such as a "take a number" approach.

The Software Dock framework provides many technologies to raise the level of abstraction for software deployment. The release and field docks themselves provide common mechanisms to access and interrogate release and field sites respectively. The registries inside the release and field docks provide high-level information through their standardized schema. These pieces provide an infrastructure that any deployment process can leverage to perform its tasks; the Software Dock does not merely drop deployment process creators out to the Turing machine, rather it provides building blocks for the process. The standard base agents are also building blocks that are useful as base classes to derive other, more specialized agents.

Although the Software Dock uses a declarative specification to support software deployment, it also recognizes the importance of procedural activities. Using the indirect event procedure call mechanism of the registry, it is possible for the Software Dock to meet the requirement for unified access to procedural resources. This requirement is particularly important for extensibility as well as any form of client-side processing that is necessary during the performance of a deployment activity.

The wide-area event mechanism, which is the heart of Software Dock connectivity, provides bi-directional, semi-continuous connectivity between the software producer and the software consumer in the Software Dock framework. Additionally, agents take full use of standard Internet protocols.

The Software Dock maintains producer and consumer autonomy through a variety of mechanisms. For example, a single software producing organization controls its own release dock. A given release dock is not affected by other release or field docks. The release dock is the only interface that other software producers and software consumers have to a given software producing organization. Other software producers can use this interface to help build their system of systems and software consumers can use this interface to browse and initiate the deployment of software releases. The field dock is a means for the software consumer to maintain autonomy. Software consumers are in control of their own field dock. Security mechanisms can limit the access that software producers have to a consumer's field dock. In addition, the consumer may also restrict or constrain operations using standard process-oriented schema elements in the field dock registry when applicable. A final piece of the Software Dock framework that supports autonomy is the wide-area event mechanism. The event mechanism is based on a publish-and-subscribe approach that does not require that a sender specifically know to whom they are sending events, nor does a receiver need to specifically know from whom they are receiving events. Thus the connectivity only requires a loose binding between the senders and receivers of events.

In order to create a unified software deployment technology, it is very important that the solution is platform independent. At a very low level, the Software Dock addresses this requirement because it is implemented entirely in Java, which is a largely platform independent programming environment. The Software Dock addresses this requirement in a conceptual fashion as well. The release and field docks create conceptual interfaces to software producer and consumer sites; any site can participate in the Software Dock framework if they can mimic these interfaces. Additionally, since a large portion of the Software Dock approach is declarative, i.e., the DSD format for describing software releases, it is possible to create many varied processes that interpret and manipulate DSD specifications.

9.3 Critical Analysis Of Related Technologies

This section critically analyzes related technologies with respect to the required software deployment characteristics presented in Section 9.1. In most cases this critical analysis is moot because these related technologies are not intended to provide a completely unified software deployment framework. Despite this fact, it is important to critically evaluate the related technologies in order to illustrate their merits and detriments with respect to software deployment and the Software Dock.

9.3.1 Traditional Configuration Management

Traditional configuration management systems, such as RCS, SCCS, Adele, and PCL, do not directly support software deployment. They do not provide abstractions for the consumer site or deployment process. Some configuration management systems, such as Adele and PCL, do provide strong software system abstractions. Since configuration management systems are not intended to support software deployment, their process coverage is low to nonexistent, they do not support deployment coordination, nor do they address any of the required capabilities except for possibly platform independence in some cases.

The real value of traditional configuration management systems with respect to deployment is their attempts to address the complexity of software systems by modeling revisions, variants, and configurations. These models enable the important, but often neglected activity of configuration selection. Particularly with Adele and PCL, which create sophisticated software system models, it is possible to describe the variability within a software system and then to select valid configurations from that description. Since these tools do not address deployment directly, though, they are not useful as deployment solutions, although it might be possible to build a deployment language on top of the general languages that they provide or to extract deployment-related information.

9.3.2 Software System Install, Update, and Remove

Software system install, update, and remove tools do not create a good, overall deployment solution. They do address their specific deployment processes reasonably well, though. They do not, however, provide a complete consumer site abstraction; most of these tools are tied to particular platforms and expose specific information that is suitable for that platform. It is not the intent of these tools to create or provide a consistent consumer site abstraction across all platforms, and thus, it is difficult to extend them to multiple platforms.

In some cases, these tools do provide reasonable software system abstractions, such as with InstallShield. These models are generally difficult to use outside of the specific tool, since they are not represented in a declarative fashion. The software system models provided by these tools are very rich with respect to file handling, but this richness does not extend to other aspects of software system modeling. For example, it is possible to select a configuration of files, but this selected configuration does not map onto the other pieces of deployment information, such as assertions and dependencies. To perform these mappings it is necessary to resort to the direct editing of a proprietary script or other programmatic trap doors.

The process coverage of these tools is generally limited to the specific process they are intended to support. Many of the install tools, such as InstallShield, PC-Install, and

WISE, do provide some support for multiple processes, but the support is limited. They support reconfiguring by re-executing the install script, but there is no automatic carry-over of configuration information, so this approach is literally a re-install rather than a reconfigure. It is possible to install patches or to install new versions over the top of existing deployed software, so in this fashion update is supported, but the lack of automatic configuration carry-over limits this approach as well.

Some tools, such as netDeploy, support update explicitly. These tools tend to take a content delivery-oriented approach, and thus suffer from the same limitations as content delivery systems. Other update tools, such as OilChange and LiveUpdate Pro, take a more manual approach to updates, where updates are simply recorded into a central database and are then scanned against the software deployed at a site. These centralized approaches do not scale and do not provide any intelligence when applying updates; thus they make no effort to guarantee consistency. There is no support for the adapt process, even though it fits easily into their models. As an aside, some newer install technologies, that are not yet released, claim to support some forms of the adapt process. There is support for removal in most of the install tools, but this support is suspect since a host of removal tools exist to remove software systems that were not properly removed. These tools do not provide any support for coordinating deployment processes.

With respect to required deployment capabilities, these tools do not address Internet scalability to any degree. In performing their specific tasks, they do raise the level of abstraction for software deployment to some degree, but they do not do this in a general, unified way and thus, when a process deviates from standard the tools provide little support. Currently these tools do not provide unified access to procedural resources, although procedural resources are available via standard programmatic methods.

Except for the tools that explicitly address the update process, there is little leveraging of the connectivity between software producers and consumers. In most cases, including the update tools, the connectivity is used merely to download artifacts and possibly for polling for new artifacts. There is no provision for feedback from consumers and little support for deployment processes to cooperate explicitly with the software release site. The notion of autonomy is lacking in these solutions. In most cases complete access to the consumer site is possible since these deployment process can execute arbitrary programs.

9.3.3 Software System and Site Description

A handful of software system and site description technologies have been introduced and these are important due to either their usefulness, their acceptance in the industry, or the significance of their proponents. A representative sample of these technologies are analyzed and evaluated in the following subsections. Due to the fact that these description technologies do not, by themselves, perform any software deployment processes, they are not strictly analyzed with respect to the deployment solution requirements presented in Section 9.1. With respect to the deployment solution requirements, these technologies only address software system abstraction and, to some degree, consumer site abstraction. A detailed evaluation of these technologies requires that they be analyzed with respect to the software system description requirements of Section 5.3.

9.3.3.1 The Open Software Description (OSD) Format

OSD, an XML application for packaging software systems, is an initial attempt by Microsoft and Marimba to create a standard for describing software systems and components for deployment. OSD is not expressly intended to address the software deployment life cycle described within this thesis, but is geared towards using "push" technologies to perform the install and update processes. Despite this modest goal, however, it is clear than many of the same requirements outlined in Section 5.3 are relevant to OSD. As it stands, only limited forms of the install and update process are possible with OSD.

Examining the OSD specification, it is clear that there are two halves to the specification: the Microsoft half and the Marimba half. Given the issues surrounding the collaboration and the time constraints involved in dealing with market forces, it appears that this initial attempt was merely Microsoft and Marimba combining their separate simple, description efforts into one standard. In particular, the Microsoft half is concerned with some consumer-side properties such as operating system type and version, while the Marimba side is simply considered a different implementation type from the Windows 95 implementation type assumed by Microsoft.

The main thrust of OSD is to describe a single revision of multiple variants of a software system. This is made possible by the standardization of a few consumer-side properties, such as operating system and computing platform. In this scenario, Java is simply treated as a variant implementation. In addition, the information provided in an OSD specification is too vague to infer required processing, other than checking the basic assertions and retrieving the necessary artifact archives. More detailed OSD evaluation is presented below. Refer to Appendix C for an example OSD specification.

Consumer site description: OSD lacks an extensive consumer site model, thus making it difficult to use assert constraints to maintain a properly working configuration of a software system. At the most basic level, OSD assumes that there is a standardized mechanism for determining a handful of consumer-side properties, such as operating system, memory, and disk space. The implicit consumer site description is too simplistic to answer sophisticated questions such as which software is dependent upon a particular property. Microsoft presumably intends to use the Registry component of Windows 95 to answer basic questions for Windows-centric platforms. Marimba is not generally concerned with these types of issues since they are concentrating on a platform neutral environment (i.e., Java). Neither approach is particularly well suited for the task at hand because the Windows Registry has no inherent schema definition for deployment and Java is not as platform neutral as one might hope.

Assert constraints: The OSD specification directly supports only a limited set of assert constraints. The specification indicates that OSD is extendable by third parties using separate XML namespaces, but this approach results in a fragmented solution to software deployment if each interested producer produces a proprietary schema. The schema must be enriched and broadened to avoid the same situation that currently exists with many software deployment tools using proprietary schema. In such a scenario, the only value that OSD provides is combining each proprietary schema into a common syntax, which is a tenuous contribution at best.

The assert constraints in OSD have limited usefulness; they are intended for the selection of an implementation and nothing more. In other words, the supported assert constraints are static consumer-side properties that are checked once during an install or update and then forgotten. The notion of using these constraints to maintain the integrity or operational correctness of a software system is not supported. Thus it is vague, for example, what affect updating a subsystem would have on already deployed dependent software systems.

Dependency constraints: OSD does support the notion of software system and component dependencies, but these are merely URL pointers to another OSD specification. The semantics of a dependency in OSD are particularly weak, see Section 9.3.3.2 of the MIF evaluation for further applicable details.

Artifacts, Configuration, and Activities: Artifact description is absent from the OSD specification. It is assumed that an OSD specification merely points to an archive or a set of archives to install. There is no support in OSD for configurable software systems. The described software system is either considered monolithic with respect to its archive or that any configuration occurs in an external install or update process. Lastly, OSD is clearly not capable of supporting the variety of activities in the software deployment life cycle. For all of these shortcomings, the solution is to fall back to the extensibility of XML namespaces to allow each producer to introduce their own notion of process description. As before, this solution is insufficient if the true intent is to provide some form of a unified software deployment framework.

9.3.3.2 The Software Management Information Format (MIF)

The Software MIF from the DMTF Application Management working group has a longer history, and hence is better defined than the OSD specification effort. The major contribution of the DMTF is the collection of schema definitions from the various working groups. Specifically related to software deployment, the Application Management working group has defined the Software MIF for describing software systems. This type of standardization is necessary to provide any type of general solution to the software deployment dilemma. Support for MIF is growing in the commercial sector, as companies are releasing computing systems that conform to various pieces of the Software MIF standard. Wide-scale adoption has still not occurred and only basic capabilities are provided such as inventory management. Currently the Software MIF only supports the install, update, and remove processes, while the adapt process receives limited support in the form of operational assert constraints.

A single Software MIF specification describes a single revision and variant of one software component. One very important aspect about the Software MIF is that it is not “Web enabled” in the sense of using any World Wide Web standards or protocols. Some of the Software MIF characteristics are detailed below. Appendix D shows a simple Software MIF specification.

Consumer site description: The complete definition of the DMTF effort specifically includes a client-side model through its DMI interface. DMTF has established working groups to create standard schema to describe various hardware configurations as well as the software description discussed here.

Assert constraints: Assert constraints are supported in the form of attribute dependencies. Asserts can constrain any attribute defined in the standard DMTF schema definitions.

Asserts are used to verify that the proper configuration of a software release is installed, although they are not used to select the proper configuration. Asserts could also possibly enforce operational correctness through the provided consumer site model, but this is not specifically mentioned nor discussed by the Software MIF.

Dependency constraints: Software MIF dependencies are not very flexible. The defined semantics of a dependency is always a specific component version or range of versions; thus, a dependency is a versioned subsystem. There is no support for dependency constraints based on a subsystem configuration or capability. For example, a software system may depend upon an optional capability of a subsystem. In this scenario, if the subsystem were deployed with the optional capability disabled, an install process for a dependent software system would succeed if it merely checked for the existence of the subsystem, rather than its precise configuration. Further, implicit in the assumption that a dependency is always a versioned subsystem, is the assumption that an installation process always resolves a dependency. The notion that it is possible to resolve a dependency by another process, such as reconfigure, is not included.

No clear support exists for the notion of a generic or capability dependency such as the generic dependency on an HTML viewer. The Equivalence group in the Software MIF schema is possibly useful here, but the degree of its usefulness is unclear since it requires the creation of an explicit list of vague equivalences. For example, a high-level declaration stating that two text editors, such as `vi` and `emacs`, are equivalent is not useful if a dependency is on a text editor with a scripting language.

Artifacts: The Software MIF provides most of the necessary support for artifact description by providing a place to specify each software artifact that comprises the software system, including its destination and type. Interestingly, the artifact descriptions do not include a source for the artifact, thus illustrating that the Software MIF is not intended for packaging software releases.

Configuration and Activities: There is no support for configurable software systems in the Software MIF. It is assumed that the component description is all-or-nothing; otherwise, any configuration is handled externally. In addition, the Software MIF is largely devoid of process or activity related description. At a very high level, the Software MIF supports install and remove processes by providing a place in the schema to point to install and remove scripts, respectively. This is a high-level process description, but it is not very useful. This approach does not allow general interpretation or reasoning about the processing steps or activities that are performed, which in turn disallows the creation of a generic procedure for handling these processes.

9.3.3.3 Other Software System and Site Description Technologies

Tivoli created AMS as an extension to the Software MIF. Thus, the entire discussion above for the Software MIF is applicable to AMS. AMS has specifically added better support for describing application topologies; in other words, it has better description support for systems of systems. The application topologies are mapped onto the business model of a particular organization, such as the activities of particular departments and groups. AMS also extends the Software MIF to describe interfaces and notifications of a software system that are relevant to managing the software system. The specialized activity description was extended to include placeholders for administrative scripts in the software description. In the end, though, AMS is still a static description of a software system that is not amenable to manipulation, much like the Software MIF that it extends.

The Defense Information Infrastructure Common Operating Environment (DII COE) notion of describing software packages as segments suffers from many of the same problems as MIF and AMS. Segment descriptions are static configuration descriptions that are not intended for manipulation. Instead, it is assumed that a central authority determines the configuration of individual software releases and then distributes them accordingly.

RPM from Redhat does not assume a central authority and is actually intended for individual consumer sites. As a result, RPM does not extend easily to groups of managed consumer sites. RPM suffers more importantly, though, in the same areas as the other approaches in this section. RPM only provides limited configuration selection and even this is incomplete; for example, the initial deployment of a software system package requires that the consumer select the variant appropriate for his or her site. The granularity of a RPM package is a complete software system; therefore, it is not possible to deal with individual elements, such as a single file artifact. RPM is also limited in the types of assertions and dependencies that it supports.

9.3.4 Network Management and System Administration

The research described in this thesis has the most in common with network management systems such as Tivoli Enterprise, HP OpenView, and Microsoft SMS. All of these systems are concerned with managing the complexities of software deployment and provide mechanisms to support many aspects of the software deployment life cycle, including install, reconfigure, update, adapt, and remove. These systems operate under the assumption that there is a central authority that is responsible for defining and maintaining a consistent definition for its managed computing sites. They perform this task well and are very robust and useful.

In most cases these system do provide abstractions for consumer sites, software systems, and to a lesser degree processes. The process coverage of these systems is good and they do provide mechanisms for coordinated deployment. They are generally not intended for deployment at Internet-scale, although they do scale to large enterprises. They follow a framework-oriented approach and therefore do raise the level of abstraction for deployment to various degrees. They leverage connectivity, but not necessarily between software producers and consumers, unless they are part of the same organization. Finally, these systems support multiple platforms reasonably well.

The main technical differences between these systems and the work presented in this thesis are the result of different worldviews. As stated above, these systems assume that there is a central authority that is responsible for the managed computing sites; this includes dictating allowable operations and exact software system configurations for a site. For example, these systems specify which version of a software release is installed on a given site. This approach assumes that a central authority controls the individual sites. This is a top-down approach where the complexity of software deployment is managed by limiting the complexity at a given individual site.

In the approach of network management systems there is little room for the concept that a computing site may exist in isolation or autonomy. To illustrate this point, network management systems are not intended to manage a single, non-networked computing site. The resources and the expertise that is required by these systems make their use in non-enterprise environments infeasible; this is particularly bothersome since most of the problems they address exist on a single site as well. Even if one argues that networks will connect all computers in the future and therefore all computers are manageable remotely, it does not follow that this model makes sense. In a diverse world, autonomy still must exist. For example, the average software consumer does not want someone else dictating how they configure their own computer.

Of course, autonomy is not the only concern. Cooperation between autonomous organizations is necessary to perform the complete software deployment life cycle. The network management approach does not include the notion of cooperation between software producers and software consumers. If a software system is purchased online, network management systems do little to leverage the connectivity between the software producer and consumer. The difficulty with network management systems is that they have eliminated the autonomy of the individual sites, while creating near organizational islands that have so much autonomy that they exhibit little cooperation with the outside world.

The above are not pure limitations of network management systems, rather it is the vision of the world to which they subscribe and thus the tools support this vision.

The Software Dock approach is bottom-up; it accepts the complexity of software systems and provides mechanisms to manage it without summarily eliminating it. Thus, each individual site can maintain its spectrum of variability with respect to software system configuration while still maintaining its manageability from either local or remote sources. It is important that individual software users as well as network administrators can install, reconfigure, update, adapt, and remove their software systems without fear of breakage or other deployment-related problems. The Software Dock accomplishes this by describing a software release as a logical entity that is manipulated in specific ways; these descriptions provide information necessary to automate or simplify most deployment activities. The result of the Software Dock approach is that configuration selection, such as optional components or precise versions, is still possible at the site level. Network management systems also use software system descriptions, but they tend to capture static configuration information.

To further pinpoint the limitations of network management systems, consider that these systems are based on MIF-derived or similar software description technologies. Through this decision, these systems inherit the inherent limitations of the MIF approach; see Section 9.3.3.2 for details of these limitations, some of which include lack of selection and configuration mechanisms, a simplistic notion of dependencies, and inadequate facilities for software system packaging.

Reviewing the other requirements for a software deployment solution, it is clear that these systems were not intended for Internet-scale. Although these systems do scale to large organization, the use of centralized databases to maintain site inventory and configuration information clearly does not scale to the Internet. The practice of centralized databases is also an intrusion on the autonomy requirement. These systems do not leverage the connectivity offered by the Internet to create opportunities for software producers and consumers to cooperate; again, these systems assume that they are both the producer and the consumer.

9.3.5 Content Delivery

Content delivery tools, such as PointCast, Castanet, NSBD, rsync and Minstrel, do not truly support software deployment, even though the technology they implement is necessary for most software deployment tasks. Despite effort by some producers of these tools to reposition themselves as general deployment solutions, they still lack the necessary technical features to provide a useful and general solution. In particular, these tools do not attempt to create a true consumer abstraction; rather, they assume a very simple abstraction or they create platform specific implementations. The software system abstraction they provide is simply a collection of files, directories, and specific changes to the site. The support they provide for process abstraction is virtually nonexistent and is largely limited to pure content delivery. As a result of their focused usage patterns, these tools do not provide reasonable process coverage; they only support a limited notion of install and update. There is little if any support for coordinated deployment.

As far as general deployment capabilities are concerned, it is unclear how well they scale given their limited use. These tools do not raise the abstraction level for deployment in any way, other than simplifying the process of getting content from one point in the network to another. These tools do leverage the connectivity that networks provide since this connectivity was the motivation for the creation of content delivery systems, but the benefits of this leveraged connectivity are inadequate for the purposes of software deployment. Finally, these system do not fully address issues of autonomy and generally provide an all-or-nothing solution.

9.3.6 Configurable Distributed Systems

Configurable distributed system (CDS) technologies are not primarily concerned with software deployment. For example, these technologies do not address general artifact delivery; instead it is assumed that artifacts are put in place by some other mechanism than the CDS technology itself. These technologies address some aspects of consumer site abstraction, but tend to assume homogenous environments. The software system abstraction is strong in CDS technologies, but they tend to treat all software systems as collections of interconnected software components. In this scenario, a component is the base unit of deployment and there is no support for configurability within the component itself. The process abstraction provided by CDS technologies is limited to the specific processes that the technologies support.

As a result of the limited process abstraction, deployment process coverage suffers. CDS technologies focus on the update, reconfigure, and adapt processes, but the focus on these processes differs from their descriptions in Section 3.1.2. CDS technologies do not focus on the actual deployment of software artifacts. Instead, they concentrate on the runtime aspects of the update, reconfigure, and adapt processes. These CDS processes specifically deal with modifying a software system while it executes, not modifying the collection of physical, on-disk artifacts that comprise the software system. One result of this emphasis on runtime issues is good support for coordinated distributed software systems. The runtime support provided by CDS is outside of the scope of software deployment as defined in this thesis.

Further examination of CDS technologies reveals that there is no support for general software release; therefore the role of the software producer is severely limited. Since the role of the software producer is lacking, issues of autonomy do not apply to CDS technologies. It is also unclear if these technologies are useful at an Internet scale, since they are intended for manipulating a specific configuration of a software system within a specific enterprise. Since these technologies do not address traditional deployment processes, they do not raise the abstraction level for deployment, although they do raise the level of abstraction for performing runtime changes to a software system.

One specific CDS system [5] proposes using agents to aid in performing reconfigure and update tasks. Additionally, this approach proposes using agents to actually deliver updated component code. Despite these similarities to the Software Dock, this approach still deals with CDS in a typical sense, and all of the issues described above in this section still apply. This approach is not intended as a general deployment framework, rather it is specifically concerned with runtime reconfigure and update of distributed software components. Further, it appears that this proposed approach was never realized.

Another particularly relevant CDS system [58] does deal with issues that are central to software deployment, but this system is still not generally applicable to software deployment. Specifically, the research admittedly does not deal with issues of installation and removal of software systems, it only supports a limited from of installation that is basically configuration copying and the only other process it supports is adapt. Additionally, the system does not support any form of configuration selection or software release. Most of the issues regarding network management systems in Section 9.3.4 apply to this system.

9.3.7 Component Models

Component models such as JavaBeans, Enterprise JavaBeans, and ActiveX are intended to facilitate interoperability among reusable modules of functionality. As such, component models are not directly related to software deployment. They do, however, overlap some aspects of software deployment. Component models provide a consumer abstraction in the form of a client-side component framework into which components are deployed; the client-side framework is generally referred to as a component container. The general applicability of this form of consumer abstraction is limited with respect to general software deployment. Software system abstraction is an integral part of component models since the sole purpose of a component model is to create a standard way to manipulate and use a software module. Of course, component models do not follow a declarative approach, but this is offset by runtime typing information available in modern programming languages, such as C++ and

Java. There is no process abstraction in component models since they provide no notion of process support; in other words, there is no direct support for the software deployment life cycle processes.

It is necessary to explicitly state that since component models are not designed as a deployment technology they do not contain enough information to describe a software system or component for deployment, as is the case of the Software Dock's DSD description format. Despite the fact that component models have standardized aspects of components and use runtime type information to enable component manipulation, this is not sufficient to gather the required information for deployment.

Also, much like CDS technologies, there is no support for general software release; therefore the role of the software producer is severely limited. Since the role of the software producer is lacking, issues of autonomy do not apply to component models. Also similarly to CDS technologies, component models are intended for creating a specific configuration of a software system within a specific enterprise; it is not clear how to meaningfully extend component models to an Internet scale. Since these technologies do not address traditional deployment processes, they do not raise the abstraction level for deployment.

One general aspect of component models does have some relevance to software deployment. The intended usage of component models is to describe the software components that comprise a given software system and the interconnections among those components. This information is a characterization of a configuration of the software system. It is possible to leverage this configuration description to automatically deploy the required software component artifacts for a given software system. Further, the JavaBeans and Enterprise JavaBeans component models also encompass an artifact packaging mechanism via Java Archive (JAR) files. The JAR files are deployable units that contain artifacts and some declarative description.

Unfortunately, trying to leverage these mechanisms to create a form of automatic deployment processing is difficult; this difficulty arises from the multiple issues. First, the component model standards do not specify how components are actually connected; the process of defining and describing the connections among components is left up to the associated builder tools. Without a standard "configuration" description language, it is very difficult to automatically recognize the software system configurations. Additionally, in cases where declarative information is available, its usefulness for deployment is very limited. For example, the declarative information in JavaBean JAR files does not include a means to specify constraints on the consumer environment, although a very limited form of dependency is supported. With respect to JAR files specifically, despite the fact that they are deployable artifacts, the information they contain is limited to actual JavaBean components and is not useful if a portion of a component is not a component itself or is native, such as with legacy systems. Lastly, component models assume that the software system configuration is monolithic; there is no notion of configurability. It is reasonable to assume, though, that component models could be extended to specifically deal with deployment issues, thus facilitating automatic extraction of deployment information for use in a deployment framework.

9.4 Usage Experiments

The three subsections that follow describe how the Software Dock prototype was applied to three different usage experiments. Each of the software systems presented in the experiments is an independent software system that was created without any special consideration regarding the Software Dock. The purpose of each experiment is to gauge the Software Dock's support for software deployment.

9.4.1 Online Learning Academy

The Online Learning Academy (OLLA) is an online interactive learning environment for students. OLLA was funded by the Defense Advanced Research Project Agency and was developed by Lockheed Martin. OLLA is characterized as a content-based system since it consists primarily of HTML and multimedia content. OLLA is a large system, consisting of 45 megabytes of content across 1700 files. The system depends on two subsystems called Disco and Harvest. Harvest is a Web page indexing system developed at the University of Colorado, while Disco is an extension to Harvest that was developed at Lockheed Martin.

The purpose of the OLLA project was to deploy and use it in military schools overseas. The deployment of OLLA was complicated by many factors. Its size and complexity made it difficult to manipulate. The schools into which OLLA was deployed did not have any computing administrative support staff, thus there was very little reliance on on-site assistance. It was also necessary to perform configuration processes at the consumer site in order to finalize the deployment of OLLA. Finally, the fact that there were numerous overseas deployment sites made the remote deployment process expensive in terms of person effort.

9.4.1.1 Existing Deployment Solution

In order to deploy OLLA, Lockheed Martin created a custom install script. The custom install script was approximately 6000 lines of Perl code that performed the extraction and installation of OLLA and its dependent subsystems, including the consumer site configuration processes. The install script was not responsible for the retrieval of any of the required software artifacts; therefore it was the responsibility of the person performing the deployment to ensure that the OLLA, Disco, and Harvest archives were transferred to the consumer site. In order to execute the installation script, the user was required to inform the script as to the whereabouts of the required archives.

The basic operation of the installation script asked the user a series of questions that were necessary for it to perform its tasks, such as questions about the local HTTP server's configuration. At certain points during the execution of the installation script, the user was directed to execute arbitrary commands by issuing the standard UNIX control-Z key sequence to break out of the executing script process. Once the commands were executed the user would then wake the sleeping installation script process to continue.

The deployment operations supported by the installation script were limited to only installation. When updates were released it was assumed that the entire OLLA configuration at the consumer site, less the dependent subsystems, was removed and the entire installation process repeated.

9.4.1.2 Software Dock Deployment Solution

The deployment solution for OLLA in the Software Dock framework required the creation of a DSD specification for the OLLA system. In order to create the DSD specification it was necessary to understand the constraints of OLLA itself, as well as its internal structure with respect to dependent subsystems. Additionally, it was necessary to understand the consumer site processes that were required in order to properly configure OLLA.

OLLA itself is a monolithic system; therefore it required very little structural planning to create its DSD specification. The only structure present in OLLA was with respect to its dependent subsystems, Disco and Harvest. As a means to demonstrate other features of DSD and the Software Dock, the OLLA DSD specification was enhanced to provide configuration options that were not fully supported by OLLA itself. For example, the DSD specification for OLLA was extended to allow the consumer to specifically select which Web content to install; the end result is the creation of some dead hyperlinks since OLLA did not actually support partitioning.

The creation of the DSD specification for OLLA first defined a set of properties to allow the configuration options. These properties mimic a standard Microsoft Windows 95-style installation process that allows a user to select full, compact, or custom deployment configurations. Besides creating properties in the DSD specification for each of these installation options, it was necessary to create properties for the specific Web content categories. For example, properties were created for science, music, art, and other groups of content available in the OLLA system.

Once the software system properties were defined in the DSD, the composition rules relating these properties to each other were created. For OLLA, the composition rules were relatively simple. The full, compact, and custom properties were defined as mutually exclusive and then the content related properties were related to the custom installation property as sub-properties.

There were no assertions in the OLLA specification since OLLA is platform independent. As mentioned above, though, there were two dependencies for OLLA in the form of the Disco and Harvest subsystems; these were included in the dependency section of the DSD specification. Specialized activities to configure and activate Harvest were added to the activity section of the DSD specification. Also, interfaces to enable the update, adapt, reconfigure, and remove processes were added to the interface section of the DSD specification. The last piece of the OLLA specification was the artifact descriptions. The DSD schema editor simplified data entry of artifact information because it is able to recursively read artifact directories from the file system. The artifacts were added to the DSD specification grouped with respect to their artifact guard conditions. For example, the music content was loaded in one group with a guard condition that required a true value for either the full install property or the music property.

Full support for OLLA required the creation of DSD specifications for its two dependent subsystems, Disco and Harvest. Disco also depends on Harvest because it is merely a simple extension to it. Since OLLA must actually execute Harvest during various deployment activities, specialized agents were created to provide indirect interfaces to Harvest via the field dock registry. These indirect interfaces enabled specialized, consumer-side processing during the OLLA deployment processes.

The resulting DSD specification for OLLA was approximately 49,000 specification lines. Although the number of specification lines is large, the current storage format for DSD specifications is particularly verbose since it is simply a text file. The number of lines that correspond to actual values that the DSD creator had to type is approximately 150 lines. These 150 lines provide a deployment solution for OLLA that is much more sophisticated than the original solution. The standard deployment agents in the Software Dock support the initial installation of OLLA and its dependent subsystems. During installation the consumer can manually configure OLLA itself and can also reconfigure it anytime after the initial installation. After initial installation, the consumer may also perform "pull" updates at anytime to verify the timeliness of their deployed configuration; "push" updates are also available if the field dock is properly configured. In the advent of improper operation of the deployed OLLA system, the consumer may use the adapt agent to verify the correctness of their deployed configuration. Finally, the consumer may remove the deployed OLLA configuration by using the standard remove agent.

9.4.2 The Chimera Open Hypermedia System

The Chimera research project [2][3] was conducted at the University of California at Irvine to explore open hypermedia systems. An open hypermedia system enables the annotation of disparate media types with arbitrary hyperlinks. Chimera was written in Java, although it has non-Java components as well. Chimera is largely divided into two halves, the server half and the client half. The server side of Chimera is a collection of related components that serve to extend a Web server to open hypermedia concepts by providing a database of open hyperlinks. The client side is a Chimera client server and a collection of Chimera client programs for viewing media that is annotated with open hyperlinks.

Besides the actual components described above, Chimera has dependent subsystems that are required for it to operate properly. In particular, the server side of Chimera is dependent upon two systems called Aelfred and Sax, which are used to provide XML parsing capabilities. The client side of Chimera is dependent upon Aelfred and Sax also, but it has an additional dependency on a system called Rivendell, which is a tool used to invoke Chimera clients. Rivendell itself has dependencies on three other systems as well. The client side of Chimera also includes five Chimera client programs that exhibit complex relationships among themselves. An emacs client program requires the Chimera shell client program, but the Chimera shell client program is also independent of the emacs client program. Additionally, a Framemaker client program has two variants, one for Microsoft Windows 95 and one for Solaris.

9.4.2.1 Existing Deployment Solution

The existing deployment solution for Chimera consisted of a Web page with pointers to the required archives for Chimera itself and its dependent subsystems. The Web page included deployment instructions as well. The entire process was performed manually.

9.4.2.2 Software Dock Deployment Solution

DSD specifications for Chimera were created that separated the server and client portions of the system itself; previously the Chimera server and client portions were distributed together since they are not resource intensive. It was not necessary to create software system properties in the DSD specification for the server portion of Chimera since it exhibited no variability. The server specification merely described file artifacts and subsystem dependencies. Since the client portion of Chimera exhibited some interesting relationships, its DSD was created to allow the consumer to configure the deployed components. In order to achieve this effect, its software specification included software properties for each of five client programs; these properties were then associated with the corresponding artifacts via guard expressions. A composition rule was created to ensure that the Chimera shell program was automatically included if the emacs client program was selected. In order to ensure that the appropriate variant of the Framemaker client program was installed depending upon the platform of the consumer site, an additional guard condition was attached to the Framemaker client program artifacts.

The standard deployment agents in the Software Dock support the initial installation of the Chimera server and client portions and their dependent subsystems. During installation of the client side of Chimera the consumer can select which components to deploy and can also reconfigure these selections anytime after the initial installation. After initial installation the consumer may perform “pull” updates at anytime to verify the timeliness of their deployed configuration; “push” updates are also available if the field dock is properly configured. In the advent of improper operation of the deployed Chimera system, the consumer may use the adapt agent to verify the correctness of their deployed configuration. Finally, the consumer may remove the deployed Chimera configurations by using the standard remove agent.

9.4.3 Java Development Kit

The Java Development Kit (JDK) from Sun Microsystems is an implementation of the Java language and programming environment. JDK is a relatively simple system that is normally distributed via the Internet. Updates to JDK are released on a regular basis. The system itself has no dependent subsystems and has four independent configuration options; these options enable the installation of program files, various header and library files, demonstration files, and source files. A small set of non-optional files is installed irrespective of the selected configuration and is referred to as the core file set.

9.4.3.1 Existing Deployment Solution

JDK is distributed in multiple formats; the format examined here is the self-extracting archive for the Microsoft Windows 95 platform; the actual installation tool used to create the self-extracting archive is the widely used InstallShield. The existing deployment solution requires that the user download a nine megabyte archive file. Once the archive is retrieved, the user executes it to start the installation process. The installation process displays information, such as a license agreement, and then allows the user to select the configuration options. Once the user has chosen the configuration, the corresponding file artifacts are installed on the consumer site.

In order to draw comparisons between the standard deployment processes and the Software Dock deployment processes for the JDK, experiments where conducted to obtain time-to-completion measurements. The experiments measured the time to install, remove, reconfigure, and update the JDK. Table 1 summarizes the results of these experiments; for detailed information on the experiments refer to Appendix E and for additional summary information refer to the following subsection.

The experiments with the standard InstallShield deployment processes for the JDK exposed interesting limitations in the default deployment process handling. The install process does not recognize existing JDK installations and this leads to multiple complications. For example, if a user installs another copy of the JDK, then any previous installations are no longer removable since the install process automatically overwrites the previous remove routine in the Microsoft Registry. Additionally, there is no real notion of reconfiguring the installed JDK, the user simply re-executes the self-extracting archive. The re-execution of the self-extracting archive does not provide any feed back to the user with respect to the location of the currently installed JDK or its precise configuration. A consequence of this lack of awareness of the deployed configuration is that a reconfiguration request to remove software components does not actually remove them. Instead, it merely installs the selected software components again and updates the remove routine for the new configuration. The result is that the software components that should have been removed are still present but they are no longer associated with the remove routine. Inefficiencies are also evident in reconfiguration requests to add software components; in this case the install process does not account for components that are already installed and simply installs them again. Finally, since JDK updates are distributed as complete, self-extracting archives too, they exhibit all the same problematic characteristics described here as well. It is undesirable to install an update "over" an existing JDK installation since the update process does not properly remove the older version of the JDK. To avoid these problems, it is better to remove the old version of the JDK before installing the new one.

These observations, however, are not necessarily true of all InstallShield deployments. InstallShield provides a full scripting language that allows many of these issues to have programmatic solutions. Therefore, these observations may result from improper or misguided usage of InstallShield, but they do illustrate an important issue: even though most software deployment processes appear simplistic, simple-minded approaches are not adequate since they rarely address the intricacies of the issues.

9.4.3.2 Software Dock Deployment Solution

The creation of the DSD specification for the JDK was straightforward because of the limited number of configurations of the JDK and the fact that each configuration was largely independent of the others. To create the DSD specification, the existing install process was executed five different times to install the five individual components, namely the core files, program files, header and library files, demonstration files, and the source files. For each installation the DSD schema editor tool was used to recursively read the associated file artifacts and to generate their corresponding schema descriptions. Since the core files were extracted each time the install was performed, care was taken not to duplicate their inclusion. Each artifact group was associated with a guard to check the value of the corresponding property used to select the JDK components except the core files, which require no guard. The entire set of artifacts was associated with a guard that corresponded to the version of the JDK, 1.1.6 in this case.

The next step was to create the actual software system properties in the property section of the DSD schema. A property representing the version of the JDK was created, as were properties to represent each of the components except the core files. Since there were no interdependencies between the properties it was not necessary to create any composition rules. Because the described JDK was for the Microsoft Windows 95 platform, an assertion was added to verify that the operating system was indeed Microsoft Windows 95.

At this point the DSD specification for version 1.1.6 of the JDK was complete. For the purposes of performing an update, the same process described above was used to extend the JDK specification to include file artifacts for JDK version 1.1.7. The newer version of JDK only changed file artifacts and did not require any structural changes to the DSD specification. The new artifacts were associated with the appropriate version and property guards. The final result was two DSD specifications, one that described version 1.1.6 of the JDK, and one that described both version 1.1.6 and version 1.1.7 of the JDK.

Through the use of these DSD specifications, the standard Software Dock agents were able to automate the install, update, adapt, reconfigure, and remove deployment processes for the JDK. The observed issues and complications of the InstallShield deployment solution, as discussed in Section 9.4.3.1, are not applicable to the Software Dock deployment processes since these issues were explicitly considered when creating the generic deployment processes.

To provide more quantitative comparisons of the Software Dock deployment solution to the existing InstallShield deployment solution for the JDK, a set of experiments to measure time-to-completion was performed. The experiments measured the time to install, remove, reconfigure, and update the JDK. Table 1 summarizes the results of these experiments; for detailed information on the experiments refer to Appendix E.

Table 1 illustrates that the prototype of the Software Dock framework is clearly a feasible approach. In the worst case, the Software Dock prototype's performance is nearly identical to InstallShield's. In all other cases the Software Dock prototype performs better than the InstallShield solution. In particular, the Software Dock prototype was significantly faster in the reconfigure experiment where software components were added. This is because

Table 1: Software Dock comparison to InstallShield.

JDK 1.1.6 Install		
	InstallShield	**Software Dock**
Transfer	15.0s	9.2s
Preparation	44.3s	26.5s
Package	n/a	53.3s
Extract	108.7s	78.2s
Overhead	n/a	4.8s
Total time:	168.0s	172.0s
JDK 1.1.6 Remove		
	InstallShield	**Software Dock**
Delete	80.0s	36.7s
Total time:	80.0s	36.7s
JDK 1.1.6 Reconfigure Remove		
	InstallShield	**Software Dock**
Preparation	42.3s	4.0s
Delete	n/a	36.3s
Extract	47.7s	n/a
Total time:	90.0s	40.3s
JDK 1.1.6 Reconfigure Add		
	InstallShield	**Software Dock**
Transfer	n/a	3.0s
Preparation	43.0s	3.8s
Package	n/a	31.2s
Extract	241.3s	66.0s
Overhead	n/a	9.3s
Total time:	284.3s	113.3s
JDK 1.1.7 Update		
	InstallShield	**Software Dock**
Transfer	13.3s	8.0s
Preparation	42.3s	33.2s
Delete	n/a	36.8s
Package	n/a	44.8s
Extract	94.0s	60.3s
Overhead	n/a	4.2s
Total time:	149.6s	187.3s

the Software Dock deployment processes are aware of the precise configuration of the installed software release and automatically deploy only the needed artifacts.

Incidentally, the comparisons between the reconfigure remove processes and the update processes are not entirely accurate comparisons. In both cases, the InstallShield solution is not actually performing the desired process. Specifically, both experiments require the deletion of existing software artifacts, but InstallShield does not delete artifacts as part of its default processing. With regards to the update experiment, the time to delete unnecessary artifacts is included in the overall completion time for the Software Dock and, thus, a direct comparison is difficult.

Additionally, there are two simple optimizations that could further improve the performance of the Software Dock prototype. First, the release dock dynamically generates artifact packages for each deployment process that requires artifacts. An extension to the release dock could cache popular configurations, thus eliminating much of the packaging time component. The second optimization deals with both the internal data structure and the on-the-wire formats used to represent the DSD specifications. These representations are very simplistic and verbose and thus contribute a great deal to the preparation time component. The creation of more terse representations or specification compression could alleviate some of this cost.

Chapter 10

Software Dock Prototype Implementation

This chapter provides details about the implementation of the Software Dock prototype. It is not intended as an exhaustive presentation of all implementation details; rather, it is intended to provide an overview of the most important implementation details. With respect to the Software Dock architecture defined in Chapter 4, all components have been implemented to some degree except the interdock.

10.1 Generic Dock

The Software Dock prototype introduces the concepts of release docks, field docks, and interdocks. These notions are derived from the generic notion of a dock. The generic dock provides a “docking” point for agents by exposing two interfaces, a registry interface and an event notification interface. These two interfaces are indicative of the central role of a typical dock, to house and manage a registry of data. Therefore, the registry in the generic dock is a critical piece of the Software Dock framework.

The structure of the registry was influenced by the structure of both MIF and the Microsoft Registry, but subsumes both. The dock registry is a hierarchical data structure where there are five primitive data types. The five primitive data types are collection, string, double, boolean, and expression. Most of these types are self-explanatory, but two, collection and expression, need further explanation.

A collection is a container of attributes. An attribute is a name-value pair, where the name is an arbitrary, unique string and the value is any legal value of one of the primitive data types. Since an attribute may have a value that is any of the primitive data types, including a collection, it is possible to build arbitrary hierarchies of nested collections. The registry itself is a wrapper around a root collection and its sub-collections.

Using the registry interface it is possible to navigate, manipulate, create, and remove attributes from the root collection of the registry. The registry adds event capabilities to the primitive collection type, thus whenever a collection or attribute is modified in any way, an event notification is generated that signals the occurrence of the specific modification. The registry then exports additional interfaces to allow interested parties to subscribe for specific event notifications. As interested parties receive event notifications, concomitant actions are performed. This general interaction is used to stimulate most of the process activity in the Software Dock framework.

The generic dock and registry infrastructure does not attach intrinsic semantics to the contents of the registry. The registry merely provides storage and structure for information that others may wish to place inside of it. It is expressly anticipated that applications of the dock and registry infrastructure will define a standard, hierarchical schema for the information that is stored in the registry in order to provide some form of semantics. This is the precise approach that the Software Dock framework has taken in the creation of the Deployable Software Description format.

10.2 Events

The registry is built around the simple notion of nested collections of artifacts. The registry, and in turn the generic dock, provides an interface to browse and manipulate the hierarchical repository of information. This static function of the registry is sufficient for storage purposes, but in the Software Dock framework, the registry is a dynamic participant that enables support for the full range of software deployment processes. To this end, the registry adds an event notification system to its hierarchical repository of information.

An event is generated each time there is a modification to the information in the registry. The types of modifications to the registry include the addition of attributes, the removal of attributes, and the update of attribute values. In response to one of these types of events, an event notification is sent to interested parties. The structure of an event notification consists of three main pieces of information: the type of the event, the registry path name associated with the event, and additional arbitrary attributes that constitute the payload of the event.

The event notification type corresponds to the action performed on the registry that generated the event, specifically attribute addition, removal, and update. Therefore, the corresponding event types are *add*, *remove,* and *update*. A fourth event type, *fire,* also exists. A fire event does not directly correspond to a registry attribute operation; it simply corresponds to a specific attribute in the registry. This mechanism is used to create communication or notification channels within the registry itself.

The event notification name corresponds to the position in the registry where the event occurred. The registry is a collection of attributes that are name-value pairs, where a value may be another collection. The nesting of collection attributes creates a hierarchy where a specific position in that hierarchy is represented as a series of names, i.e., a pathname. Therefore a pathname is the concatenation of the attribute names leading to a given point in the tree, separated by a path separator character; this is identical to the approach used in most file systems to identify file paths.

The final element in the event notification is the event attribute list; the event attribute list does not necessarily correspond directly to the event that occurred. Instead, the event attribute list is used to pass an arbitrary payload to interested parties. In order to attach these arbitrary attributes to an event, all registry interfaces that generate an event also accept an optional attribute list to attach to the resulting event notification. This mechanism further enables communication channels within the registry by passing additional information to interested parties.

The registry provides an event subscription interface to allow interested parties to receive events. The event subscription interface is relatively simple, allowing interested parties to specify the type and the name of events in which they are interested; the type and the name directly correspond to the event type and event name described above. An interested party must also supply a callback routine to receive any dispatched event notifications. The subscription mechanism extends the notion of an event name slightly to create an event name specification. An event name specification extends pathnames to include wildcard characters. There are two types of wildcard characters in an event name specification. The "." character matches any attribute name at a single level in the pathname, while the "*" character matches any number of attribute names across zero or more levels in the pathname. The purpose of the event name specification is to allow interested parties to subscribe for entire classes of events without knowing the specific pathnames in advance.

The meaning of an event, by definition, indicates the occurrence of a specific modification to the registry. As with the generic dock and registry infrastructure described above, there is no attempt to attach any other intrinsic semantics to an event. It is expressly anticipated that applications of the dock and registry infrastructure will place additional meaning on the occurrence of events. Through the use of a standard registry schema, applications can interpret an event based on a higher level understanding of the standard registry schema. For example, in a deployment specific schema, the addition of a child attribute in the registry node "/Local/Software/Applications" could indicate that a new application was added to the site.

Events can also mimic remote or cross-address space procedure calls. To understand this mechanism, consider that a collection in the registry is actually a queue where each attribute in the queue is uniquely identified by its name. The addition of an attribute to a collection is viewed as a procedure request and the removal of the added attribute is viewed as an end to the request. A standard schema may define certain registry nodes as interfaces. An interested party that services requests sent to an interface simply subscribes for all attribute addition events, i.e., a request. When a request arrives, the interested party responsible for processing the request receives a notification and may then perform any implied processing. When the processing is complete, the party performing the request must de-queue the attribute that represented the request in order to signal that the request has finished. The party that originated the request can subscribe for the remove event to receive notification of its completion. Using the event attribute list, it is possible to pass additional parameters into an interface, as well as allow an interface to return values to the requester.

The immediate nature of events complicates software systems, such as the Software Dock, that use them to stimulate processing. In an effort to deal with this issue, the registry introduces the notion of transaction-oriented events. The purpose of transaction-oriented events is to limit the visibility of an event until the task it is associated with is complete. This concept was presented in [54] and has gained importance since many distributed systems are pursuing event-based approaches. Transaction-oriented events are necessary in event-based systems when concomitant actions are performed upon the receipt of an event. Specifically, assume that an attribute is inserted into the registry and arbitrary child attributes are added to that attribute. Now assume that an interested party performs an action in response to the addition of the first attribute and uses the child attributes as parameters. If the interested party responds too quickly to the addition event of the first attribute, all of the child attributes may not yet exist. In such a scenario it is important that the originator of the events is done inserting all of the attributes before actions are performed in response to them. Transaction-oriented events limit the visibility of events in this fashion.

The transaction-oriented events provided by the registry do not imply true transactional semantics since they are not durable with respect to the state of the registry. For example, aborting an event transaction does not undo the changes made to the registry. The transaction-oriented events simply limit visibility by allowing a party using the registry to indicate that the events that it is creating are part of a logical unit. To do this, the party using the registry creates a registry task to indicate the start of a transaction. Event notifications that are generated by registry operations are queued on the task until the party using the registry indicates that the task is complete. At the completion of a task, the registry dispatches all event notifications to interested parties. It is also possible to abort a task, at which point the registry simply drops all queued event notifications.

A task may also have sub-tasks that have the same semantics as the top-level task, except that when a sub-task completes, its events are only delivered to its parent task or sub-task. This implies that an event subscription also has an associated event task that allows participants in the task to receive events as they occur within the task or when sub-tasks complete. Event notifications dispatched from a sub-task are queued in the parent task in an upward fashion until the top-level task completes. Interested parties that are not participants in a given task may only receive events when the top-level task completes. The exception to this rule is when an interested party subscribes for events in "immediate" mode. An immediate subscription will receive an event as soon as it occurs. In this case, the interested party can join the associated task so that any subsequent events that it generates are logically associated with the task in question and thus limited in visibility.

10.3 Release Dock

Release docks are surrogates for software producers in the Software Dock framework. The purpose of a release dock is to manage the software releases that a software producer wishes to make available to software consumers via a network, such as the Internet. To achieve this goal, the release dock provides interfaces to browse and retrieve its software releases.

The release dock implementation is a derivative of the generic dock. As such, a release dock contains a registry for storing hierarchical information and an event notification mechanism that signals changes in its registry. The release dock introduces a simple, standard schema in its registry to allow interested parties to browse and manipulate it. The simplest characterization of the release dock registry schema is that it describes the collection of software releases available at a given release dock. The description of each software release is relatively minimal, only indicating the name of the software family for each release. In order to gain specific knowledge about a given software release, an interested party must use the other interfaces provided by the release dock itself. In addition to the generic dock interfaces, the release dock provides interfaces for requesting agents, retrieving a family schema, creating a package, submitting a release, and updating a release.

The agent request interface allows other agents and Software Dock infrastructure pieces to request specialized software deployment processes from the release dock; a more detailed discussion of agents is deferred until Section 10.5. The family schema retrieval interface allows an interested party to request the DSD specification for a software release; the list of available releases is determined by examining the registry of the release dock. An alternative solution is to physically include the DSD specification in the release dock registry, thus eliminating the need for this interface, but this approach is generally not as efficient since it requires many remote accesses to the release dock registry. In the current approach the DSD specification is sent in one request. Additionally, certain operations, such as updating a software release's DSD specification in the release dock, are more complex if the alternative approach is pursued.

The package creation interface of the release dock allows interested parties to request the creation of an artifact package, such as a ZIP file. A party that requests a DSD specification for a given software release will manipulate that specification to arrive at a list of artifacts that it needs to perform some deployment task. Using artifact signatures from within the DSD specification, it is possible to request a specific set of artifacts for a software release. The package creation interface simply returns a URL to the dynamically created package.

The above interfaces are used by or on the behalf of software consumers. There are also interfaces for the software producer that owns the release dock to submit and update software releases. These interfaces allow a software producer to manipulate the content of their release dock by submitting new or updated DSD specifications. These interfaces also perform another vital function; they create events that represent new or updated software releases at the release dock. These events are then dispatched to interested parties to perform concomitant actions.

The release dock implementation is currently not a database in its own right. It is assumed that DSD specifications point to artifacts that are under external control. It is possible and reasonable that alternative implementations of the release dock may choose to make internal copies of all artifacts, which requires dynamic update of DSD specifications with respect to artifact locations when a new or updated release is submitted.

Up until this point the described release dock interfaces are programmatic in nature. The release dock is also responsible for making software releases visible to the software consumer using more conventional mechanisms, namely a Web browser. As software releases are added and updated, the release dock dynamically generates Web pages using the software release's DSD specification. These Web pages are used by software consumers to view and instigate the installation of software releases.

10.4 The Field Dock

A field dock is a surrogate for software consumer in the Software Dock framework. The purpose of a field dock is to house information that describes the state of the consumer's site as well as providing an interface to the site. The information in the field dock is used as context from which deployment processes are derived.

The field dock implementation, like the release dock's, is a derivative of the generic dock. Therefore, the field dock also contains a registry for storing hierarchical information and an event notification mechanism to signal changes in the registry. The field dock also introduces a standard schema in its registry to allow interested parties to browse and manipulate it. The current standard field dock registry schema contains information about the computing platform, operating system, interfaces, and software releases deployed at a given consumer site. In addition to the generic dock interfaces, the field dock provides an interface for retrieving an artifact package from a release dock for subsequent manipulation and an interface to gain access to the directory of a deployed software release.

The interface for retrieving an artifact package is used in conjunction with the release dock's interface to create an artifact package. When performing a software deployment activity at a field site, it is possible to request a package of artifacts from a remote release dock and then use the URL returned by the release dock to request the local field dock to retrieve the artifact package. The local field dock retrieves the artifact package and returns a package reader object to the requester. The package reader gives access to the contents of the package via an interface that reads the contents of the artifact package.

Most deployment activities require some access to the local file system. The field dock provides limited access through its file system interface. This interface returns an object that provides access to the root directory of a specific software release. Given the registry path to a specific software release in the field dock, the file system access interface returns an application writer object. The application writer provides normal file system access routines, but only for the root directory where the software system is installed. As such, any file system manipulation, such as creating or deleting files, is limited to the software release's directory tree.

The relationship between a DSD specification and the field dock's standard registry schema is also interesting. As stated above, the schema in the field dock registry includes information about the deployed software releases at the field site. The field dock does not create its own schema for describing software releases; instead, it uses the DSD specification of a software release as its description. The field dock schema does not incorporate a DSD specification verbatim as its description for a software release; the configuration of the deployed software system is also recorded.

In order to record the configuration information, the DSD specification is extended slightly in the field dock registry. The extensions to the DSD specification include additional collections to record the configuration's final property values, valid interfaces, and valid notifications. This extended DSD specification is referred to as an instantiated DSD specification. The reason for this terminology is due to the fact that a DSD specification describes many valid configurations of a software release, whereas an instantiated DSD specification describes a specific instance of a valid configuration.

The field dock itself is very simple, it provides only two primary interfaces. These interfaces were chosen because they are necessary for nearly every deployment operation. It is possible that other interfaces may eventually be added to the field dock as experience indicates, but the field dock interface is not intended to change rapidly. Instead, the combination of using the event mechanisms of the registry and a standard registry schema to create arbitrary event interfaces is the preferred method to extend the field dock. This mechanism, as described in Section 10.2, enables the creation of arbitrary procedures that are similar to remote procedure calls. Interfaces created with the registry event mechanism do not require any physical changes to the field dock, nor is the field dock even aware of them. It is important not to underestimate the value of this mechanism since it is currently used in the prototype to invoke all deployment processes, as well as arbitrary activities that are required during the deployment of specific software releases.

10.5 Agents

In the Software Dock framework, agents are used to perform most software deployment functionality. Agents perform this functionality by using the interfaces provided by the release and field docks, and by interpreting the DSD specification and consumer site information. Therefore, the agent embodies a generic algorithm that is parameterized by the information in the deployment environment.

An agent has very few specific characteristics; it is simply a unit of code that is used to distribute deployment process functionality. The infrastructure of the Software Dock implementation lends itself to this task; it was created with Java, which provides a common computing platform across heterogeneous sites, and with Voyager [43], an object request broker, which provides inter-process communication capabilities and simple code mobility. These infrastructure pieces allow any object to take on agent-like qualities, especially with regard to code mobility. Thus, an agent has no tangible definition other than it is a unit of code that is possibly mobile and works on behalf of either the software producer, the software consumer, or both to perform a specific task. The mobility of agents is not necessarily important for distributing the collection of standard generic agents, but customized agents require code mobility for distribution.

To understand the internal workings of an agent, this section describes the details of a generic agent using pseudo-code. The pseudo-code is broken into code fragments to ease description; concatenating each code fragment forms the complete pseudo-code agent. The pseudo-code agent implements each of the five main deployment processes in the Software Dock prototype; the actual deployment process that the agent performs is determined by a parameter. The purpose of illustrating the five deployment processes as one generic agent is to demonstrate the commonality among the deployment processes; this approach does not reflect the actual agent implementations in the Software Dock prototype. Error handling is largely ignored in the pseudo-code, but a transactional notion is introduced in order to indicate the critical boundary for error handling.

10.5.1 Agent Initialization

All Software Dock agents implement a standard docking interface that is called when an agent arrives at a dock. The standard docking interface acts as an initialization routine for the agent. Besides normal variable initialization, the docking function is where an agent subscribes for necessary events and adds any necessary nodes to the local field dock registry. In general an agent adds a node to the local field dock registry in order to provide a means to accept event requests. When the install process is performed, the agent must execute a couple additional steps when it docks. The install agent must retrieve a copy of the DSD specification for the software release from the release dock, add the DSD specification to the field dock registry, and request a remove agent from the release dock. After an agent docks it becomes dormant and simply waits for a deployment process request event.

```
public class Agent {
   private int process = -1;
   private String releaseDockAddress = null;
   private String fieldDockAddress = null;
   private String familyName = null;
   private Properties boundProperties = null;

   private String dockPoint = null;
   private FieldDock fielddock = null;
   private ReleaseDock releasedock = null;

   public int dock(int process, String releaseDockAddress, String fieldDockAddress,
      String familyName, Properties boundProperties)
   {
      this.process = process;
      this.releaseDockAddress = releaseDockAddress;
      this.fieldDockAddress = fieldDockAddress;
      this.familyName = familyName;
      this.boundProperties = boundProperties;

      fielddock = Namespace.lookup(fieldDockAddress);
      dockPoint = fielddock.getDockingPoint();

      if (process == INSTALL)
         fielddock.subscribe(dockPoint + "/Configuration/Interfaces/Install/.");
      else if (process == RECONFIGURE)
         fielddock.subscribe(dockPoint + "/Configuration/Interfaces/Reconfigure/.");
      else if (process == UPDATE)
         fielddock.subscribe(dockPoint + "/Configuration/Interfaces/Update/.");
      else if (process == ADAPT)
         fielddock.subscribe(dockPoint + "/Configuration/Interfaces/Adapt/.");
      else if (process == REMOVE)
         fielddock.subscribe(dockPoint + "/Configuration/Interfaces/Remove/.");

      if (process != REMOVE)
         fielddock.subscribe(dockPoint + "/Configuration/Notifications/Removed");

      releasedock = Namespace.lookup(releaseDockAddress);

      if (process == INSTALL)
      {
         Family family = releasedock.retrieveFamilySchema(familyName);
         fielddock.addFamilyInstance(dockPoint, family);
         releasedock.requestAgent("RemoveAgent", familyName, fieldDockAddress,
            dockPoint);
      }

      return 0;
   }
```

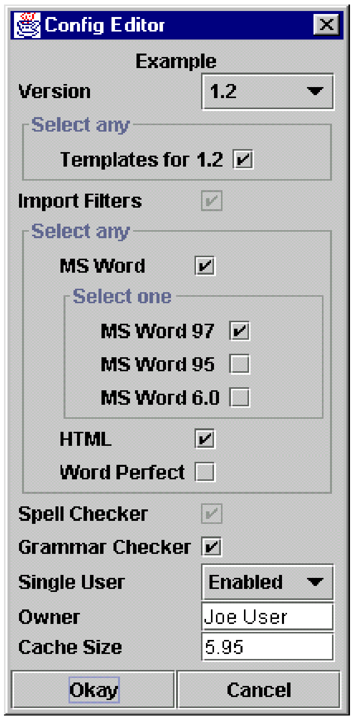
10.5.2 Event Dispatch and Process Activation

The event mechanism of the generic dock is used to stimulate a deployment process request. The agent's event dispatching function is invoked when an event is received from the field dock. In general, deployment agents receive either a deployment process request or a remove notification. A remove notification signals to the agent that its associated software family instance has been removed and, thus, the agent should shut down. A deployment process request causes the agent to perform its associated deployment process.

The agent starts the deployment process by displaying a progress window to inform the user of its progress. The deployment processes must determine the current configuration of the software family instance as well as the target configuration. It is valid to have a null current configuration, such as install, and it is also valid to have a null target configuration, such as adapt and remove. The adapt process's target configuration is null because it does not change the software family configuration it merely enforces the current configuration. The current software family configuration is determined by examining the schema instance in the field dock registry; the target configuration is determined either by bound parameters that are passed into the agent or by querying the user with the configuration editor [see Figure 4].

The configuration editor uses the DSD specification to determine the software family properties and how the user can manipulate those properties. The composition rules from the DSD specification ensure that a valid configuration is selected by dynamically constraining the software family's property values while the user manipulates them.

Once a valid target configuration is determined, the


Config Editor

Example

Version: 1.2

Select any
- Templates for 1.2 [✓]

Import Filters [✓]

Select any
- MS Word [✓]
  - Select one
    - MS Word 97 [✓]
    - MS Word 95 [ ]
    - MS Word 6.0 [ ]
- HTML [✓]
- Word Perfect [ ]

Spell Checker [✓]

Grammar Checker [✓]

Single User: Enabled

Owner: Joe User

Cache Size: 5.95

Figure 4: The Configuration Editor.

agent builds applicable lists of the various DSD specification elements for both the current and the target family configurations. To build these lists the agent uses the property values of both configurations to test the guard conditions of the associated schema elements; when a guard returns true then the schema element is applicable to the given configuration. The result of this process is lists of applicable assertions, dependencies, interfaces, notifications, artifacts, and activities for both the current and target configuration. The agent then processes each schema element list.

```
public void dispatchEvents(Event event)
   {
      if ((process == INSTALL) && (event.getPath().startsWith(dockPoint +
          "/Configuration/Interfaces/Install/")))
         performProcess(event);
      else if ((process == RECONFIGURE) && (event.getPath().startsWith(dockPoint +
          "/Configuration/Interfaces/Reconfigure/")))
         performProcess(event);
      else if ((process == UPDATE) && (event.getPath().startsWith(dockPoint +
          "/Configuration/Interfaces/Update/")))
         performProcess(event);
      else if ((process == ADAPT) && (event.getPath().startsWith(dockPoint +
          "/Configuration/Interfaces/Adapt/")))
         performProcess(event);
      else if ((process == REMOVE) && (event.getPath().startsWith(dockPoint +
          "/Configuration/Interfaces/Remove/")))
         performProcess(event);
      else if (event.getPath().equals(dockPoint +
          "/Configuration/Notifications/Removed"))
      {
         if (event.getAttribute("Result") == SUCCESS)
         {
            if (process == INSTALL)
               fielddock.unsubscribe(dockPoint +
                  "/Configuration/Interfaces/Install/.");
            else if (process == RECONFIGURE)
               fielddock.unsubscribe(dockPoint +
                  "/Configuration/Interfaces/Reconfigure/.");
            else if (process == UPDATE)
               fielddock.unsubscribe(dockPoint +
                  "/Configuration/Interfaces/Update/.");
            else if (process == ADAPT)
               fielddock.unsubscribe(dockPoint +
                  "/Configuration/Interfaces/Adapt/.");

            fielddock.unsubscribe(dockPoint +
               "/Configuration/Notifications/Removed");

            die();
         }\
      }
   }

   public void performProcess(Event event)
   {
      int ret = ERROR;

      ProgressWindow progressWindow = new ProgressWindow();
      progressWindow.show();

      Family currentFamily = null;
      Family targetFamily = null;
      Properties currentConfig = null;
      Properties targetConfig = null;

      if ((process == RECONFIGURE) || (process == UPDATE) ||
          (process == ADAPT) || (process == REMOVE))
      {
         currentFamily = fielddock.getFamilyInstance(dockPoint);
         currentConfig = currentFamily.getConfiguration();
      }

      if ((process == INSTALL) || (process == RECONFIGURE))
         targetFamily = fielddock.getFamilyInstance(dockPoint);
      else if (process == UPDATE)
         targetFamily = releasedock.retrieveFamilySchema(familyName);

      if ((process == INSTALL) || (process == RECONFIGURE) || (process == UPDATE))
         targetConfig = configure(targetFamily);

      Assertions newAssertions = oldAssertions = null;
      Dependencies newDependencies = oldDependencies = null;
      Activities newActivities = oldActivities = null;
      Interfaces newInterfaces = oldInterfaces = null;
      Notifications newNotes = oldNotes = null;
      Artifacts newArtifacts = oldArtifacts = null;
      ConfigVariableBinder targetBinder = currentBinder = null;

      if (currentConfig != null)
      {
         currentBinder = new ConfigVariableBinder(fielddock, currentConfig);

         oldAssertions = buildList(Assertions, Assertion,
            currentFamily.getAttributeValue("Assertions"), currentBinder);
         oldDependencies = buildList(Dependencies, Dependency,
            currentFamily.getAttributeValue("Dependencies"), currentBinder);
         oldActivities = buildList(Activities, Activity,
            currentFamily.getAttributeValue("Activities"), currentBinder);
         oldInterfaces = buildList(Interfaces, Interface,
            currentFamily.getAttributeValue("Interfaces"), currentBinder);
         oldNotes = buildList(Notifications, Notification,
            targetFamily.getAttributeValue("Notifications"), currentBinder);
         oldArtifacts = buildList(Artifacts, Artifact,
            currentFamily.getAttributeValue("Artifact"), currentBinder);
      }

      if (targetConfig != null)
      {
         targetBinder = new ConfigVariableBinder(fielddock, targetConfig);

         newAssertions = buildList(Assertions, Assertion,
            targetFamily.getAttributeValue("Assertions"), targetBinder);
         newDependencies = buildList(Dependencies, Dependency,
            targetFamily.getAttributeValue("Dependencies"), targetBinder);
         newActivities = buildList(Activities, Activity,
            targetFamily.getAttributeValue("Activities"), targetBinder);
         newInterfaces = buildList(Interfaces, Interface,
            targetFamily.getAttributeValue("Interfaces"), targetBinder);
         newNotes = buildList(Notifications, Notification,
            targetFamily.getAttributeValue("Notifications"), targetBinder);
         newArtifacts = buildList(Artifacts, Artifact,
            argetFamily.getAttributeValue("Artifact"), targetBinder);
      }

      beginTransaction();

      if (process == ADAPT)
         ret = processAssertions(newAssertions, oldAssertions, currentBinder);
      else
         ret = processAssertions(newAssertions, oldAssertions, targetBinder);

      if (ret == SUCCESS)
```

```
      {
         if (process == ADAPT)
            ret = processDependencies(newDependencies, oldDependencies,
               currentBinder);
         else
            ret = processDependencies(newDependencies, oldDependencies, targetBinder);

         if (ret == SUCCESS)
         {
            if ((processActivities(newActivies, oldActivities) == SUCCESS) &&
                (processInterfaces(newInterfaces, oldInterfaces) == SUCCESS) &&
                (processNotifications(newNotes, oldNotes) == SUCCESS) &&
                (processArtifacts(newArtifacts, oldArtifacts) == SUCCESS))
            {
               if (process == REMOVE)
                  fielddock.removeFamilyInstance(dockPoint);
               else if ((process == INSTALL) || (process == RECONFIGURE) ||
                        (process == UPDATE))
                  fielddock.updateFamilyInstance(dockPoint, targetConfig);
            }
            else
               ret = ERROR;
         }
      }

      if (ret == SUCCESS)
         commitTransaction();
      else
         abortTransaction();

      result(event, ret);
   }

   protected void result(Event event, int result)
   {
      progressWindow.close();

      fielddock.removeAttribute(event.getPath());

      if (process == INSTALL)
         fielddock.generateAttributeEvent(dockPoint +
            "/Configuration/Notifications/Installed", result);
      else if (process == UPDATE)
         fielddock.generateAttributeEvent(dockPoint +
            "/Configuration/Notifications/Updated", result);
      else if (process == RECONFIGURE)
         fielddock.generateAttributeEvent(dockPoint +
            "/Configuration/Notifications/Reconfigured", result);
      else if (process == ADAPT)
         fielddock.generateAttributeEvent(dockPoint +
            "/Configuration/Notifications/Adapted", result);
      else if (process == REMOVE)
         fielddock.generateAttributeEvent(dockPoint +
            "/Configuration/Notifications/Removed", result);

      if (process == INSTALL)
      {
         if (ret == SUCCESS)
         {
            fielddock.unsubscribe(dockPoint + "/Configuration/Interfaces/Install/.");
            fielddock.unsubscribe(dockPoint + "/Configuration/Notifications/Removed");
            die();
         }
         else
            fielddock.addAttribute(dockPoint + "/Configuration/Interfaces/Remove",
               "RemoveRequest");
      }
      else if (process == REMOVE)
      {
         if (ret == SUCCESS)
         {
            fielddock.unsubscribe(dockPoint + "/Configuration/Interfaces/Remove/.");
            die();
         }
      }
   }

   protected Collection buildList(Type collectionType, Type elementType,
      Collection collection, ConfigVariableBinder binder)
   {
      Collection applicable = null;

      if (collection.getGuard().evaluate(binder) == TRUE)
      {
         applicable = new Collection();

         for (int i = 0; i < collection.size(); i++)
         {
            if (collection.get(i) instanceof collectionType)
            {
               Collection temp = buildList(collectionType, elementType,
                  collection.get(i), binder);

               if (temp != null)
                  applicable.append(temp);
            }
            else if (collection.get(i) instanceof elementType)
            {
               if (collection.get(i).getGuard().evaluate(binder) == TRUE)
                  applicable.append(collection.get(i));
            }
         }
      }

      return applicable;
   }

   protected Properties configure(Family family)
   {
      ConfigManager cm = new ConfigManager(fielddock, family);

      for (int i = 0; i < boundProperties.size(); i++)
      {
         cm.setPropertyValue(boundProperties.get(i).getName(),
            boundProperties.get(i).getValue());
         cm.setPropertyReadOnly(boundProperties.get(i).getName(), true);
      }

      if (cm.showConfigurationDialog() == OKAY)
         return cm.getConfiguration();

      return null;
   }
```

10.5.3 Assertion Processing

The agent processes applicable assertions by evaluating the assert condition of each of the applicable assertions with respect to the current property values and the current field dock registry. If any assertion is false, the agent displays the violated assertion's description to the user and returns a failure result. If all assertions are verified without a failure, then a success result is returned. Assertions are processed in a slightly different manner than the other schema elements. Since assertions are merely test conditions, there is no associated “undo” action when transforming the current configuration to the target configuration. Therefore, the assertions for the current configuration are generally ignored. The exception to this rule is when the adapt process is performed, in this case the current assertions are simply re-evaluated.

```
protected int processAssertions(Assertions newAssertions, Assertions oldAssertions,
   ConfigVariableBinder binder)
{
   if (process == REMOVE)
      return SUCCESS;

   if (newAsserions == null)
      newAssertions = oldAssertions;

   for (int i = 0; i < newAssertions.size(); i++)
   {
      if (newAssertions.get(i).evaluate(binder) == FALSE)
      {
         showMessageDialog("Assertion Failed",
            newAssertions.get(i).getDescription());
         return ERROR;
      }
   }

   return SUCCESS;
}
```

10.5.4 Dependency Processing

In the current Software Dock prototype, a dependency always represents a dependent subsystem, although this will change in the future. The agent processes the dependencies in a two-step fashion. First, all old dependencies are removed if they are not also in the new dependency list, thus removing the actual subsystems. Next, all new dependencies are added if they are not in the old dependency list, thus adding the actual subsystems. This two-step process is referred to as differential processing because the agent performs a do/undo process on the difference of the two lists. The agent resolves a dependency by requesting an agent, as described in the DSD specification, to come and perform the necessary resolution routine. The exception to the process describe here is the adapt process; the adapt process simply re-tests and then re-resolves any old dependencies that are broken.

```
protected int processDependencies(Dependencies newDependencies,
   Dependencies oldDependencies, ConfigVariableBinder binder)
{
   Dependencies addDependencies = null;

   if (process == ADAPT)
   {
      for (int i = 0; i < oldDependencies.size(); )
      {
         if (oldDependencies.get(i).evaluate(binder) == TRUE)
            oldDependencies.remove(i);
         else
            i++;
      }

      newDependencies = oldDependencies;
      oldDependencies = null;
   }

   if (oldDependencies != null)
   {
      Dependencies removeDependencies = (newDependencies == null) ?
         oldDependencies : oldDependencies.subtract(newDependencies);
      addDependencies = (newDependencies == null) ? null :
         newDependencies.subtract(oldDependencies);

      for (int i = 0; i < removeDependencies.size(); i++)
      {
         fielddock.addAttribute(
            removeDependencies.get(i).getRemoveInterfaceName(), "InvokeRequest");

         if (waitForAgentToFinish() == ERROR)
            return ERROR;
      }
   }
   else
      addDependencies = newDependencies;

   for (int i = 0; (addDependencies != null) && (i < addDependencies.size()); i++)
   {
      if (addDependencies.get(i).evaluate(binder) == FALSE)
      {
         Properties contraints = addDependencies.get(i).getConstraints();
         releasedock.requestAgent(addDependencies.get(i).getAgentName(),
            addDependencies.get(i).getFamilyName(), fielddock, constraints);

         waitForAgentToDock();

         fielddock.addAttribute(addDependencies.get(i).getInterfaceName(),
            "InvokeRequest");

         if (waitForAgentToFinish() == ERROR)
            return ERROR;
      }
   }

   return SUCCESS;
}
```

10.5.5 Activity and Interface Processing

The processing of activities and interfaces are very similar to each other. The agent performs differential processing over the old and the new lists of schema elements. To process activities or interfaces, the deployment agent requests another agent to perform the functionality associated with each activity or interface. The associated "undo" action when processing the old activities or interfaces is to remove the appropriate agents that were previously requested. In the case of the adapt process, the presence of each agent is verified and any missing agents are requested again.

```
protected int processActivities(Activities newActivities, Activities oldActivities)
{
   Activities addActivities = null;

   if (process == ADAPT)
   {
      for (int i = 0; i < oldActivities.size(); )
      {
         if (fielddock.isAgentDocked(
             oldActivities.get(i).getAgentName(), dockPoint))
            oldActivities.remove(i);
         else
            i++;
      }

      newActivities = oldActivities;
      oldActivities = null;
   }

   if (oldActivities != null)
   {
      Activities removeActivities = (newActivities == null) ?
         oldActivities : oldActivities.subtract(newActivities);
      addActivities = (newActivities == null) ? null :
         newActivities.subtract(oldActivities);

      for (int i = 0; i < removeActivities.size(); i++)
         fielddock.undockAgent(removeActivities.get(i).getAgentName(), familyName,
            dockPoint);
   }
   else
      addActivities = newActivities;

   for (int i = 0; (addActivities != null) && (i < addActivities.size()); i++)
      releasedock.requestAgent(addActivities.get(i).getAgentName(), familyName,
         fieldDockAddress, dockPoint);

   return SUCCESS;
}

protected int processInterfaces(Interfaces newInterfaces, Interfaces oldInterfaces)
{
   Interfaces addInterfaces = null;

   if (process == ADAPT)
   {
      for (int i = 0; i < oldInterfaces.size(); )
      {
         if (fielddock.isAgentDocked(oldInterfaces.get(i).getAgentName(),
             dockPoint))
            oldInterfaces.remove(i);
         else
             i++;
      }

      newInterfaces = oldInterfaces;
      oldInterfaces = null;
   }

   if (oldInterfaces != null)
   {
      Interfaces removeInterfaces = (newIntefaces == null) ?
         oldInterfaces : oldInterfaces.subtract(newInterfaces);
      addInterfaces = (newInterfaces == null) ? null :
         newInterfaces.subtract(oldInterfaces);
      for (int i = 0; i < removeInterfaces.size(); i++)
         fielddock.undockAgent(removeInterfaces.get(i).getAgentName(), familyName,
            dockPoint);
   }
   else
      addInterfaces = newInterfaces;

   for (int i = 0; (addInterfaces != null) && (i < addInterfaces.size()); i++)
       releasedock.requestAgent(addInterfaces.get(i).getAgentName(), familyName,
          fieldDockAddress, dockPoint);

   return SUCCESS;
}
```

10.5.6 Notification Processing

The agent processes notifications by adding or removing nodes from the family instance contained in the field dock registry; this is done via differential processing. In the case of the adapt process, if any old notifications are not present in the family instance, they are simply added again.

```
protected int processNotifications(Notifications newNotifications,
   Notifications oldNotifications)
{
   if (process == REMOVE)
      return SUCCESS;

   Notifications addNotifications = null;

   if (process == ADAPT)
   {
      for (int i = 0; i < oldNotifications.size(); )
      {
         if (fielddock.doesAttributeExist(dockPoint +
             "/Configuration/Notifications/" + oldNotifications.get(i).getName()))
            oldNotifications.remove(i);
         else
             i++;
      }

      newNotifications = oldNotifications;
      oldNotifications = null;
   }

   if (oldNotifications != null)
   {
      Interfaces removeNotifications = (newNotifications == null) ?
         oldNotifications : oldNotifications.subtract(newNotifications);
      addNotifications = (newNotifications == null) ? null :
         newNotifications.subtract(oldNotifications);

      for (int i = 0; i < removeNotifications.size(); i++)
         fielddock.removeAttribute(dockPoint + "/Configuration/Notifications/" +
            removeNotifications.get(i).getName());
   }
   else
      addNotifications = newNotifications;

   for (int i = 0; (addNotifications != null) &&
        (i < addNotifications.size()); i++)
      fielddock.addAttribute(dockPoint + "/Configuration/Notifications/" +
         addNotifications.get(i).getName());
```

```
    return SUCCESS;
}
```

10.5.7 Artifact Processing

The agent uses the same differential processing algorithm to process the artifact elements of the DSD specification. First, the agent removes any old artifacts that are not in the new artifact list and then it adds any new artifacts that are not in the old artifact list. In order to add the new artifacts, the agent must request an artifact package from the release dock. The release dock dynamically creates an artifact package per the agent's request and returns a URL pointer to the artifact package. The agent uses this URL pointer to make a request to the local field dock to retrieve the artifact package and unpack it. In the case of the adapt process, the old artifacts are verified by comparing their content signature from the DSD specification to the computed content signature. Artifacts are requested and installed for any signatures that do not match.

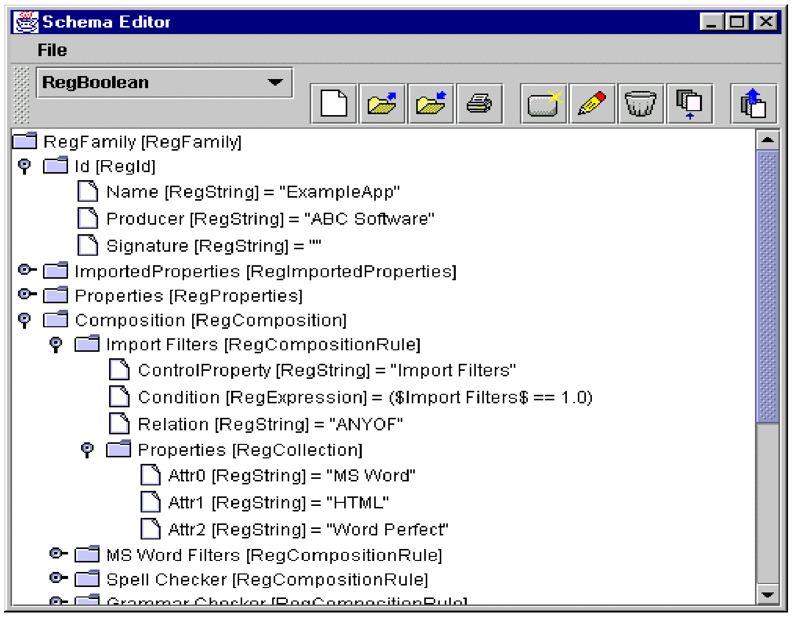
```
protected int processArtifacts(Artifacts newArtifacts, Artifacts oldArtifacts)
{
    Artifacts addArtifacts = null;

    if (process == ADAPT)
    {
        ApplicationWriter appWriter = fielddock.createApplicationWriter(dockPoint);

        for (int i = 0; i < oldArtifacts.size(); )
        {
            if (appWriter.verifyFile(oldArtifacts.getSignature()) == TRUE)
                oldArtifacts.remove(i);
            else
                i++;
        }

        appWriter.close();

        newArtifacts = oldArtifacts;
        oldArtifacts = null;
    }

    if (oldArtifacts != null)
    {
        Artifacts removeArtifacts = (newArtifacts == null) ?
            oldArtifacts : oldArtifacts.subtract(newArtifacts);
        addArtifacts = (newArtifacts == null) ? null :
            newArtifacts.subtract(oldArtifacts);

        ApplicationWriter appWriter = fielddock.createApplicationWriter(dockPoint);

        for (int i = 0; i < removeArtifacts.size(); i++)
            appWriter.deleteFile(removeArtifacts.get(i).getDestination());

        appWriter.close();
    }
    else
```

```
         addArtifacts = newArtifacts;

      if (addArtifacts != null)
      {
         String urlString = releasedock.createPackage(addArtifacts);

         if (urlString != null)
         {
            PackageReader packageReader = fielddock.retrievePackage(urlString);
            ApplicationWriter appWriter =
               fielddock.createApplicationWriter(dockPoint);

            for (int i = 0; i < addArtifacts.size(); i++)
            {
               packageReader.openEntry(addArtifacts.get(i).getSignature());
               appWriter.openOutputFile(addArtifacts.get(i).getDestination());

               for ( ; pacakgeReader.isEOF() != true; )
                  iApp.writeFile(packageReader.read(BUFSIZE), BUFSIZE);

               appWriter.closeOutputFile();
               packageReader.closeEntry();
            }

            packageReader.close();
            appWriter.close();
         }
      }

      return 0;
   }
}
```

10.6 Schema Editor and Docking Station

In order to facilitate usage and experimentation of the Software Dock framework, some specific support tools were implemented. Since the Software Dock framework relies very heavily on hierarchical specifications for both the specifications of software systems as well as computing sites, it was necessary to simplify the creation of these hierarchical specifications. The Schema Editor [see Figure 5] is a simple editor for structured hierarchical data. The Schema Editor understands the collection-based type system used in the generic dock registry. It uses this type system to create DSD specifications and computing site specifications. Besides acting as a structured editor, the Schema Editor has additional functionality to simplify some routine tasks, such as automatically reading file artifacts from the file system.

The Schema Editor is further integrated into the Software Dock framework by playing a role in the software release process. Once a DSD specification is created, the Schema Editor can submit the new or updated specification to the local release dock. Submitting a

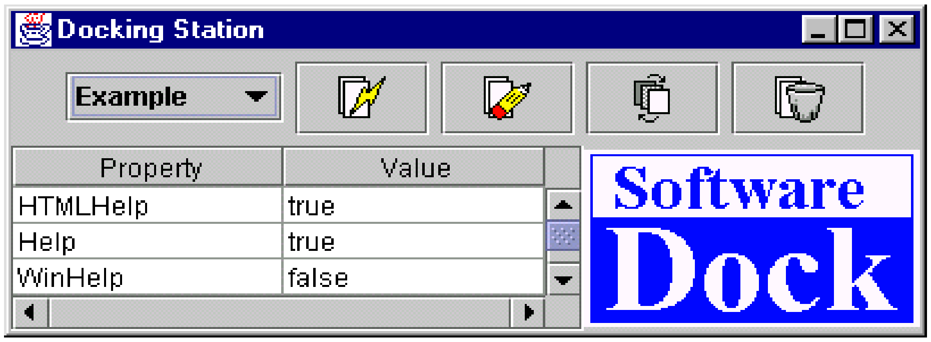


```
RegFamily [RegFamily]
  Id [RegId]
      Name [RegString] = "ExampleApp"
      Producer [RegString] = "ABC Software"
      Signature [RegString] = ""
  ImportedProperties [RegImportedProperties]
  Properties [RegProperties]
  Composition [RegComposition]
    Import Filters [RegCompositionRule]
        ControlProperty [RegString] = "Import Filters"
        Condition [RegExpression] = ($Import Filters$ == 1.0)
        Relation [RegString] = "ANYOF"
      Properties [RegCollection]
          Attr0 [RegString] = "MS Word"
          Attr1 [RegString] = "HTML"
          Attr2 [RegString] = "Word Perfect"
    MS Word Filters [RegCompositionRule]
    Spell Checker [RegCompositionRule]
    Grammar Checker [RegCompositionRule]
```

Figure 5: The DSD Schema Editor.

specification makes the software release described by the specification visible and available from the release dock.

Another utility, called the Docking Station [see Figure 6], plays an important role in accessing and demonstrating the Software Dock framework's functionality. The Docking Station is a simple tool that acts as an interface to the deployment processes of the software releases that are deployed at a given consumer site. The Docking Station determines the software releases and the deployment processes supported by those software releases by querying the field dock registry. It presents a graphical user interface to allow the user to activate specific deployment processes, such as reconfigure, for specific deployed software sys-



Property	Value
HTMLHelp	true
Help	true
WinHelp	false

Figure 6: The Software Dock Docking Station.

tems; it provides this functionality by using the event notification mechanism described in Section 10.2.

Chapter 11

Future Research

Software deployment is a broad area with many interesting research avenues to pursue. Chapter 8 listed some specific areas that were considered out of scope for this thesis, specifically security, electronic commerce, error recovery, and deployment process interactivity. These specific areas provide many research opportunities and require further investigation. Of these areas, error recovery and deployment process interactivity are of primary concern as the Software Dock framework is extended to better support the role of an administrative software consumer.

The administrative software consumer is a special type of software consumer that is responsible for managing a group of consumer sites; this person is typically referred to as a system administrator. Another server, called the interdock, is introduced to explore support for the administrator role. The interdock provides a mechanism for grouping logically related consumer sites; in this scenario, each field dock has an interdock associated with it. Any number of interdocks may exist inside of a software consumer organization, where the interdocks themselves form a hierarchy. From an administrative point of view, interdocks create logical topologies of their consumer sites, thus creating a global view of their computing environment.

The logical topologies of consumer sites are used to perform administrative tasks, such as inventorying software configurations or performing various deployment processes on groups of consumer sites. In some instances, support for these administrative tasks require the creation of new generic agents, such as an inventory agent. In order to support the standard software deployment processes, new remote deployment agents are needed. These remote agents function in a fashion similar to the current collection of deployment agents, but they are further parameterized by consumer site topologies, thus performing their deployment processes on groups of remote sites.

These remote deployment processes create interesting challenges in the configuration process since the administrator that is configuring the software system is no longer located at the remote site and the groups of remote sites might be heterogeneous. As a result, it is necessary to extend the configuration process to either require that the administrator specifically follow all possible configuration paths or to first pre-fetch relevant information from the target sites and then have the administrator follow only the necessary configuration paths. The interdock acts as a common repository in this scenario, where information for the administrative job is stored and the results are reported.

The global view of the computing environment provided by the interdock is leveraged to better support deployment of distributed systems. As a simple example, the service elements of the DSD schema provide valuable global information that is stored in the interdock. In a client/server scenario, the client is rarely deployed on the same site as the server, thus to ensure the proper deployment of the client software system it is necessary for the deployment process to resolve the service dependency. To verify that a service exists, a deployment process simply walks the interdock hierarchy. If the service exists in the interdock hierarchy then the deployment process may proceed; otherwise the deployment process may either fail or it may request a remote deployment process to install the service.

Further extension of the interdock is also possible to enable it to act as a halfway house for software releases. In such a scenario, an administrator requests a software family from a release dock and stores the software family at an interdock. The administrator most likely prunes the requested software family so that it requires fewer resources as the complete family description at the release dock. For example, the administrator might only request the last two revisions of the software family for the Microsoft Windows 95 platform. In order to take advantage of these software release halfway houses, the current software deployment processes only need to walk the interdock hierarchy requesting a software release from the local interdocks before they make a request to their release dock. This approach allows administrators to service their sites more quickly since data is moved closer to its final destination. This approach also helps organizations that are behind firewalls, where direct Internet connection is prohibited or discouraged. The end goal of the interdock is to provide centralized management while still allowing individual site autonomy.

Another area of future research is that of policy decisions. The current DSD schema has minimal support for policy decisions. In a single site consumer organization the current model is fine since the administrator and the actual consumer are one in the same. Once the roles of administrator and consumer diverge, it is then necessary to have more sophisticated policies. Currently the Software Dock prototype allows an administrator to disable certain deployment activities by simply removing their interfaces from the field dock registries, but more sophisticated control of when and how deployment processes are performed is necessary. Given the current level of information provided in the DSD schema, flexible policy support is possible for the full spectrum of control, from very loose to very strict. This is because the Software Dock framework is able to manage the variability that may exist at a consumer site since the DSD schema provides information about how a software system is manipulated. It is a complex research issue to determine the degree to which it is possible to define a purely generic agent with respect to policy decisions. At the very least, standard or common policy parameters for each deployment process require investigation.

Of all the software deployment processes in the software deployment life cycle, only two are not addressed by the Software Dock prototype at all; these are activation and deactivation. Initial support for simply executing the entry point artifacts of a software release is a possible extension, but it is not the primary concern of these processes. If a software release merely needs an artifact executed, the operating system or the graphical user shell of the consumer site is most likely sufficient. The primary area of investigation for the activation and deactivation processes is for systems that require complex startup or shutdown procedures, such as those of distributed systems. In order to support these issues it may require extensions to the DSD schema if it is not possible to map these requirements onto the generic notion of a dependency. The interdock may also play a significant role in the area of activation and deactivation. There is also some contention that full support for activation and deactivation moves beyond the scope of software deployment. A move towards runtime monitoring and maintenance may push the deployment framework too far into the network management domain or the distributed application framework domain.

Lastly, the DSD specifications for software releases provide a wealth of information for various analyses. The use of analysis tools could simplify the creation of DSD specifications themselves. Since a single DSD specification for a single revision and variant can become arbitrarily complex, creating a DSD specification that describes an entire family may become ungainly in some situations. Analysis tools could allow DSD specification creators to write DSD specifications for single revisions and variants and then automatically merge them into a family description. This approach is beneficial for simpler systems as well, since less time is required for simple but time-consuming issues, such as locating duplicate files among revisions and variants. Analysis of DSD specifications is also useful for testing the validity of the specification. For example, analysis of a DSD specification could determine whether any software system properties are unreachable or whether any other schema elements are unreachable.

Chapter 12

Conclusions

Despite the fact that software deployment issues have existed since the creation of the first software system, there is still no unified or complete framework to support them. The increased complexity of modern software systems has exacerbated the need for a unified software deployment framework. Software systems composed from multiple systems or components exhibit complex dependency patterns that must be understood and managed for proper deployment. Software deployment is further complicated by distributed systems that exhibit complex dependencies among remote as well as local software systems.

Many software deployment tools and approaches exist that address specific pieces of the software deployment problem space. Existing systems and approaches, such as traditional configuration management and network administration, all provide necessary pieces for software deployment, but none provide a complete solution. It is necessary to leverage the efforts in these related areas in order to provide a complete, unified solution for software deployment.

This dissertation's contribution to the area of software deployment has four facets. First, software deployment has been defined as a life cycle of interrelated processes. This definition and the definition of the individual processes are necessary in order to define the scope of the software deployment problem space. Second, a complete software deployment framework called the Software Dock has been defined and prototyped. The implementation of the Software Dock framework has provided valuable insight for determining the feasibility of using agents, events, and declarative schema for supporting software deployment. Third, a declarative schema for describing the deployment requirements of software systems has been created, called the Deployable Software Description (DSD) format. DSD exhibits unique characteristics with respect to other software deployment description technologies because it specifically addresses the variability of software systems themselves. By describing software system variability, it is possible to use DSD to create generic processes that can manipulate arbitrary software systems for the purpose of software deployment. Lastly, as a result of DSD and the Software Dock prototype, a generic deployment process algorithm has been defined that illustrates the relationships and commonality among software deployment processes.

References

1. B. Agnew, C. Hofmeister, and J. Purtilo. "Planning for Change: A Reconfiguration Language for Distributed Systems," Proceedings of the Second International Workshop on Configurable Distributed Systems, IEEE Computer Society, March 1994, pp.15-22.

2. K. M. Anderson. "Integrating Open Hypermedia Systems with the World Wide Web," Proceedings of the Eighth ACM Conference on Hypertext, April 1997, pp. 157-166.

3. K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr. "Chimera: Hypertext for Heterogeneous Software Environments," Proceedings of the Sixth ACM Conference on Hypertext, September 1994, pp. 94-107.

4. E. C. Bailey. "Maximum RPM," Red Hat Software, Inc., ISBN: 1-888172-78-9, Feb. 1997.

5. J. Berghoff, O. Drobnik, A. Lingnau, and C. Mönch. "Agent-based Configuration Management of Distributed Applications," Proceedings of the Third International Conference on Configurable Distributed Systems, IEEE Computer Society, May 1996, pp. 52-59.

6. C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. "A Dynamic Reconfiguration Service for CORBA," Proceedings of the Fourth International Conference on Configurable Distributed Systems, IEE Computer Society, May 1998, pp. 35-42.

7. T. Bray. "Extensible Markup Language (XML): Part I. Syntax," Textuality, Vancouver, BC, Canada. http://www.w3.org/pub/WWW/TR/WD-xml-lang.html.

8. N. Brown and C. Kindel. "Distributed Component Object Model Protocol – DCOM/1.0," Internet Engineering Task Force, Network Working Group, Internet Draft, January 1998.

9. 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.

10. 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.

11. Desktop Management Task Force. "Desktop Management Interface Specification," Version 2.0s, June 24, 1998.

12. Desktop Management Task Force. "Common Information Model Specification," Version 2.0, March 3, 1998.

13. Desktop Management Task Force. "Software Standard Groups Definition", Version 2.0, Mar. 27, 1996.

14. Desktop Management Task Force. "Enabling your product for manageability with MIF files," November 1994.

15. C. Ellerman. "Channel Definition Format", Microsoft Corporation, 1997.

16. J. Estublier and R. Casallas. "The Adele Configuration Manager," Configuration Management, Wiley, 1994, pp. 99-134.

17. H. Fosså and M. Sloman. "Implementing Interactive Configuration Management for Distributed Systems," Proceedings of the Third International Conference on Configurable Distributed Systems, IEEE Computer Society, May 1996, pp. 44-51.

18. N. Freed and N. Borenstein. "Multipurpose Internet Mail Extensions (MIME) Part one: Format of Internet Message Bodies," RFC 2045, November 1996.

19. 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.

20. 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.

21. R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. "An architecture for Post-Development Configuration Management in a Wide-Area Network," Proceedings of the International Conference on Distributed Configurable Systems, IEEE Computer Society, May 1997, pp. 269-278.

22. M. Hauswirth. "Minstrel Fact Sheet," Distributed Systems Group, Technical University of Vienna, 1998.

23. Hewlett-Packard. "HP OpenView Family Guide," 1998. http://www.openview.hp.com.

24. A. van der Hoek, R. S. Hall, D. Heimbigner, and A. L. Wolf. "Software Release Management," Proceedings of the 6th European Software Engineering Conference, September 1997.

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

26. J. Honeycutt. "Using the Windows 95 Registry," Que Publishing, Indianapolis, IN, 1996.

27. InstallShield Corp. "InstallShield," 1998. http://www.installshield.com/ispro.

28. InstallShield Corp. "InstallFromTheWeb Product Guide," Version 2.0, 1997. http://www.installshield.com/iftw.

29. 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/baselin3.pdf

30. 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.

31. T. Lindholm and F. Yellin. “The Java Virtual Machine Specification,” Addison-Wesley, 1997.

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

33. D. Mackenzie, R. McGrath, and N. Friedman. “Autoconf: Generating Automatic Configuration Scripts,” Free Software Foundation, April 1994.

34. J. Magee, N. Dulay, and J. Kramer. “A Constructive Development Environment for Parallel and Distributed Programs,” Proceedings of the Second International Workshop on Configurable Distributed Systems, IEEE Computer Society, March 1994, pp. 4-14.

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

36. McAfee Software. “OilChange,” 1999. http://www.mcafee.com/cybermedia.

37. McAfee Software. “Uninstaller,” 1999. http://www.mcafee.com/cybermedia.

38. S. Menon and R. LeBlanc, Jr. “Object Replacement using Dynamic Proxy Updates,” Proceedings of the Second International Workshop on Configurable Distributed Systems, IEEE Computer Society, March 1994, pp. 82-91.

39. Microsoft Corporation. “Zero Administration Initiative,” 1998. http://www.microsoft.com/windows/innovation.

40. Microsoft Corporation. “Systems Management Server Version 2.0, Reviewer's Guide,” 1998.

41. J. P. Mueller. “ActiveX from the Ground Up,” Osborne/McGraw-Hill, ISBN 0-07-882264-5, 1997.

42. Object Management Group. “The Common Object Request Broker: Architecture and Specification,” Revision 2.2, February 1998.

43. ObjectSpace, Inc. “ObjectSpace Voyager Core Technology 2.0 User Guide,” 1998. http://www.objectspace.com.

44. Open Software Associates. “OpenWEB netDeploy,” 1998. http://www.osa.com.

45. P. Oreizy and R. N. Taylor. "On the Role of Software Architectures in Runtime System Reconfiguration," Proceedings of the Fourth International Conference on Configurable Distributed System, IEEE Computer Society, May 1998, pp. 61-70.

46. F. Plášil, D. Bálek, and R. Janeček. "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating," Proceedings of the Fourth International Conference on Configurable Distributed Systems, IEEE Computer Society, May 1998, pp. 43-51.

47. Pointcast, Inc. "Pointcast," 1998. http://www.pointcast.com.

48. M. J. Rochkind. "The Source Code Control System," IEEE Transactions on Software Engineering, Volume SE-1, Number 4, December 1975, pp. 364-370.

49. D. Rogerson. "Inside COM," Microsoft Press, ISBN 1-57231-349-8, 1997.

50. S. K. Shrivastava and S. M. Wheater. "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications," Proceedings of the Fourth International Conference on Configurable Distributed Systems, IEEE Computer Society, May 1998, pp.10-17.

51. Sun Microsystems. "JavaBeans API Specification," Version 1.01, July 24, 1997.

52. Sun Microsystems. "Enterprise JavaBeans Architecture Specification," Version 1.0, March 21, 1998.

53. Sun Microsystems. "Java Remote Method Invocation Specification," Revision 1.50, JDK 1.2, October 1998.

54. S. M. Sutton, Jr. and L. J. Osterweil. "APPL/A: A Language for Software Process Programming," ACM Transactions on Software Engineering, Volume 4, Number 3, July 1995, pp. 221-286.

55. Symantec Corporation. "Norton CleanSweep 4.5," 1999. http://www.symantec.com.

56. Symantec Corporation. "LiveUpdate Pro," 1998. http://www.nortonweb.com.

57. Symantec Corporation. "TuneUp.com," Defunct. Transitioned into Norton Web Services, 1999. http://www.nortonweb.com.

58. J. D. Thornton. "Practical Description of Configurations for Distributed Systems Management," Proceedings of the Third International Conference on Configurable Distributed Systems, IEEE Computer Society, May 1996, pp. 36-43.

59. W. F. Tichy. "RCS, A System for Version Control," Software – Practice and Experience, Volume 15, Number 7, July 1985, pp. 637-654.

60. Tivoli Systems. "Tivoli Enterprise Overview," 1998. http://www.tivoli.com/o_products/html/enterprise36.html.

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

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

63. E. Tryggeseth, B. Gulla, and 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.

64. Twenty Twenty Software. "PC-Install for Window Version 6 Manual," 1997. http://www.twenty.com.

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

66. Wise Solutions, Inc. "Wise Installation System," 1998. http://www.wisesolutions.com.

67. M. Zimmermann and O. Drobnik. "Specification and Implementation of Configurable Distributed Applications," Proceedings of the Second International Workshop on Configurable Distributed Systems, IEEE Computer Society, March 1994, pp. 23-34.

Appendix A

Deployable Software Description Document Type Definition

The DSD specification presented here is an application of the Extensible Markup Language (XML) [7], called a Document Type Definition (DTD). The DTD for DSD describes the proper syntax for all DSD schema elements. This DTD for the DSD represents a base-level syntax for describing software systems for deployment; actual implementations of DSD may extend this base syntax for proprietary purposes, as is the case with the Software Dock's implementation of DSD.

```
<?xml encoding="US-ASCII"?>

<!ELEMENT Family (Id, ImportedProperties, Properties, Composition,
    Assertions, Dependencies, Artifacts, Interfaces, Notifications,
    Services, Activities)>

<!ELEMENT Guard (#PCDATA)>

<!ELEMENT Id (IdName, IdDescription, IdProducer, (IdLicense)?,
    IdSignature)>
<!ELEMENT IdName (#PCDATA)>
<!ELEMENT IdDescription (#PCDATA)>
<!ELEMENT IdProducer (#PCDATA)>
<!ELEMENT IdLicense (#PCDATA)>
<!ELEMENT IdSignature (#PCDATA)>

<!ELEMENT ImportedProperties (ImportedProperty)*>
<!ELEMENT ImportedProperty (PropertyName, PropertyType,
    PropertyDescription, PropertyValue)>
<!ELEMENT PropertyName (#PCDATA)>
<!ELEMENT PropertyType EMPTY>
<!ATTLIST PropertyType Value (string | boolean | double) #REQUIRED>
<!ELEMENT PropertyDescription (#PCDATA)>
<!ELEMENT PropertyValue (#PCDATA)>

<!ELEMENT Properties (Property)*>
<!ELEMENT Property (PropertyName, PropertyType, PropertyDescription,
    PropertyDefaultValue, PropertyDefaultEnabled,
    PropertyDefaultDisabled, PropertyTopLevel, PropertyValues)>
<!ELEMENT PropertyDefaultValue (#PCDATA)>
<!ELEMENT PropertyDefaultEnabled (#PCDATA)>
<!ELEMENT PropertyDefaultDisabled (#PCDATA)>
<!ELEMENT PropertyTopLevel EMPTY>
<!ATTLIST PropertyTopLevel Value (true | false) #REQUIRED>
<!ELEMENT PropertyValues (PropertyValue)*>

<!ELEMENT Composition (CompositionRule)*>
<!ELEMENT CompositionRule (RuleCondition, RuleControlProperty,
    RuleRelation, RuleProperties)>
<!ELEMENT RuleCondition (#PCDATA)>
<!ELEMENT RuleControlProperty (PropertyName)?>
<!ELEMENT RuleRelation EMPTY>
<!ATTLIST RuleRelation Value (anyof | oneof | excludes | includes )
    #REQUIRED>
<!ELEMENT RuleProperties (PropertyName)*>

<!ELEMENT Assertions (Guard, (Assertions | Assertion)*)>
<!ELEMENT Assertion (Guard, AssertionCondition,
    AssertionDescription)>
<!ELEMENT AssertionCondition (#PCDATA)>
<!ELEMENT AssertionDescription (#PCDATA)>

<!ELEMENT Dependencies (Guard, (Dependencies | Dependency)*)>
<!ELEMENT Dependency (Guard, DependencyCondition,
    DependencyDescription, DependencyResolution,
    DependencyConstraints)>
<!ELEMENT DependencyCondition (#PCDATA)>
<!ELEMENT DependencyDescription (#PCDATA)>
<!ELEMENT DependencyResolution (#PCDATA)>
<!ELEMENT DependencyConstraints (ImportedProperty)*>

<!ELEMENT Artifacts (Guard, (Artifacts | Artifact)*)>
<!ELEMENT Artifact (Guard, ArtifactSignature, ArtifactType,
    ArtifactSourceName, ArtifactSource, ArtifactDestinationName,
    ArtifactDestination, ArtifactEntryPoint, ArtifactMutable,
    ArtifactPermission)>
<!ELEMENT ArtifactSignature (#PCDATA)>
<!ELEMENT ArtifactType (#PCDATA)>
<!ELEMENT ArtifactSourceName (#PCDATA)>
<!ELEMENT ArtifactSource (#PCDATA)>
<!ELEMENT ArtifactDestinationName (#PCDATA)>
<!ELEMENT ArtifactDestination (#PCDATA)>
<!ELEMENT ArtifactEntryPoint EMPTY>
<!ATTLIST ArtifactEntryPoint Value (true | false) #REQUIRED>
<!ELEMENT ArtifactMutable EMPTY>
<!ATTLIST ArtifactMutable Value (true | false) #REQUIRED>
<!ELEMENT ArtifactPermission (#PCDATA)>

<!ELEMENT Interfaces (Guard, (Interfaces | Interface)*)>
<!ELEMENT Interface (Guard, InterfaceName, InterfaceDescription)>
<!ELEMENT InterfaceName (#PCDATA)>
<!ELEMENT InterfaceDescription (#PCDATA)>

<!ELEMENT Notifications (Guard, (Notifications | Notification)*)>
<!ELEMENT Notification (Guard, NotificationName,
    NotificationDescription)>
<!ELEMENT NotificationName (#PCDATA)>
<!ELEMENT NotificationDescription (#PCDATA)>

<!ELEMENT Services (Guard, (Services | Service)*)>
<!ELEMENT Service (Guard, ServiceName, ServiceDescription)>
<!ELEMENT ServiceName (#PCDATA)>
<!ELEMENT ServiceDescription (#PCDATA)>

<!ELEMENT Activities (Guard, (Activities | Activity)*)>
<!ELEMENT Activity (Guard, ActivityAction, ActivityWhen,
    ActivityDescription)>
<!ELEMENT ActivityAction (#PCDATA)>
<!ELEMENT ActivityWhen (#PCDATA)>
<!ELEMENT ActivityDescription (#PCDATA)>
```

Appendix B

Deployable Software Description Example

This appendix introduces an example DSD specification. This example follows the lead of the OSD specification [25] by using a fictitious implementation of Solitaire. The example illustrates most of the main capabilities of the DSD format. In particular, it illustrates support for multiple revisions and variants of the Solitaire system for Microsoft Windows 95 and for the Java Virtual Machine. As a specific note, this DSD specification is able to easily share artifacts across revisions and variants; for example, the HTML version of the on-line documentation is used for both variants and the help files themselves are shared across revisions of the system. A non-shared, external dependency on a component that provides the card images is specified for the Java implementation, although shared dependencies are perfectly valid. Finally, the ability of DSD to describe the internal variability of a software system is illustrated by the introduction of properties that enable the selection of the specific system implementation as well as the inclusion or exclusion of the on-line help documentation.

This example follows the Software Dock prototype's implementation of DSD and therefore is an extension of the standard DSD DTD designed specifically to work within the Software Dock framework. The Software Dock extends the DSD by adding parameters to dependencies, interfaces, and activities that specifically deal with release docks and agents. Additionally, consumer site property names, and expressions are all specific to the Software Dock implementation.

```
<?xml version="1.0"?>
<!DOCTYPE Family SYSTEM "dock.dtd">

<Family>
  <Id>
    <IdName>Solitaire</IdName>
    <IdDescription>The game of Solitaire</IdDescription>
    <IdProducer>FooBar Corporation</IdProducer>
    <IdLicense>http://www.foobar.com/license.html</IdLicense>
    <IdSignature>f2a3b8c32091fd46a785c39723e289a9</IdSignature>
  </Id>
  <ImportedProperties>
    <ImportedProperty>
      <PropertyName>OS</PropertyName>
      <PropertyType Value="string"></PropertyType>
      <PropertyDescription>Operating system</PropertyDescription>
      <PropertyValue></PropertyValue><!-- recorded at deployment -->
    </ImportedProperty>
  </ImportedProperties>
  <Properties>
    <Property>
      <PropertyName>Implementation</PropertyName>
      <PropertyType Value="string"></PropertyType>
      <PropertyDescription>
        Selects implementation
      </PropertyDescription>
      <PropertyDefaultValue>Java</PropertyDefaultValue>
      <PropertyDefaultEnabled>Native</PropertyDefaultEnabled>
      <PropertyDefaultDisabled>Java</PropertyDefaultDisabled>
      <PropertyTopLevel Value="true"></PropertyTopLevel>
      <PropertyValues>
        <PropertyValue>Native</PropertyValue>
        <PropertyValue>Java</PropertyValue>
      </PropertyValues>
    </Property>
    <Property>
      <PropertyName>Online Help</PropertyName>
      <PropertyType Value="boolean"></PropertyType>
      <PropertyDescription>
        Includes online help documentation
      </PropertyDescription>
      <PropertyDefaultValue>true</PropertyDefaultValue>
      <PropertyDefaultEnabled></PropertyDefaultEnabled>
      <PropertyDefaultDisabled></PropertyDefaultDisabled>
      <PropertyTopLevel Value="true"></PropertyTopLevel>
      <PropertyValues></PropertyValues>
    </Property>
    <Property>
      <PropertyName>WinHelp</PropertyName>
      <PropertyType Value="boolean"></PropertyType>
      <PropertyDescription>
        Includes WinHelp documentation
      </PropertyDescription>
      <PropertyDefaultValue>false</PropertyDefaultValue>
      <PropertyDefaultEnabled></PropertyDefaultEnabled>
      <PropertyDefaultDisabled></PropertyDefaultDisabled>
      <PropertyTopLevel Value="false"></PropertyTopLevel>
      <PropertyValues></PropertyValues>
    </Property>
    <Property>
      <PropertyName>HtmlHelp</PropertyName>
      <PropertyType Value="boolean"></PropertyType>
      <PropertyDescription>
        Includes HTML documentation
      </PropertyDescription>
      <PropertyDefaultValue>true</PropertyDefaultValue>
      <PropertyDefaultEnabled></PropertyDefaultEnabled>
      <PropertyDefaultDisabled></PropertyDefaultDisabled>
      <PropertyTopLevel Value="false"></PropertyTopLevel>
      <PropertyValues></PropertyValues>
    </Property>
  </Properties>
  <Composition>
    <!-- This rule forces the "Implementation" property to be
         Java if the operating system is not a Windows variant -->
    <CompositionRule>
      <RuleCondition>
        (($OS$ != "Win95") AND ($OS$ != "WinNT"))
      </RuleCondition>
      <RuleControlProperty></RuleControlProperty>
      <RuleRelation Value="excludes"></RuleRelation>
      <RuleProperties>
        <PropertyName>Implementation</PropertyName>
      </RuleProperties>
    </CompositionRule>
    <!-- This rule indicates that if the "Online Help" property is
         selected, then either the "WinHelp" or "HtmlHelp" must be
         selected -->
    <CompositionRule>
      <RuleCondition>($Online Help$ == true)</RuleCondition>
      <RuleControlProperty>
        <PropertyName>Online Help</PropertyName>
      </RuleControlProperty>
      <RuleRelation Value="oneof"></RuleRelation>
      <RuleProperties>
        <PropertyName>WinHelp</PropertyName>
        <PropertyName>HtmlHelp</PropertyName>
      </RuleProperties>
    </CompositionRule>
    <!-- This rule forces the "WinHelp" property to be false if the
         operating system is not a Windows variant -->
    <CompositionRule>
      <RuleCondition>
        (($OS$ != "Win95") AND ($OS$ != "WinNT"))
      </RuleCondition>
      <RuleControlProperty></RuleControlProperty>
      <RuleRelation Value="excludes"></RuleRelation>
      <RuleProperties>
        <PropertyName>WinHelp</PropertyName>
      </RuleProperties>
    </CompositionRule>
  </Composition>
  <Assertions>
    <Guard></Guard>
    <Assertion>
      <Guard></Guard>
      <AssertionCondition>
        (($OS$=="Win95") || ($OS$=="WinNT") ||
         ($OS$=="Solaris") || ($OS$=="Linux"))
      </AssertionCondition>
      <AssertionDescription>
        Only supported on Win95, WinNT, Solaris, and Linux.
      </AssertionDescription>
    </Assertion>
  </Assertions>
  <Dependencies>
    <Guard></Guard>
    <Dependency>
      <Guard>($Implementation$ == "Java")</Guard>
      <DependencyCondition>
        (!installed("Cards"))
      </DependencyCondition>
      <DependencyDescription>
        Java version depends on deck of cards component.
      </DependencyDescription>
      <DependencyReleaseDock>www.cards.com</DependencyReleaseDock>
      <DependencyFamily>Cards</DependencyFamily>
      <DependencyAgent>Install</DependencyAgent>
      <DependencyInterface>Install</DependencyInterface>
      <DependencyResolution></DependencyResolution>
      <DependencyConstraints></DependencyConstraints>
    </Dependency>
  </Dependencies>
  <Artifacts>
    <Guard></Guard>
    <Artifact>
      <Guard>($Implementation$ == "Native")</Guard>
      <ArtifactSignature>
        5d929a2486c67e71f9231697119452d0
      </ArtifactSignature>
      <ArtifactType>EXECUTABLE</ArtifactType>
      <ArtifactSourceName>solitaire.exe</ArtifactSourceName>
      <ArtifactSource>/solitaire/win/bin</ArtifactSource>
      <ArtifactDestinationName>
        solitaire.exe
      </ArtifactDestinationName>
      <ArtifactDestination>bin</ArtifactDestination>
      <ArtifactEntryPoint Value="true"></ArtifactEntryPoint>
      <ArtifactMutable Value="false"></ArtifactMutable>
      <ArtifactPermission></ArtifactPermission>
    </Artifact>
    <Artifact>
      <Guard>($Implementation$ == "Java")</Guard>
      <ArtifactSignature>
        96429c4a616df82513e45050ac03d55e
      </ArtifactSignature>
      <ArtifactType>CLASSFILE</ArtifactType>
      <ArtifactSourceName>solitaire.class</ArtifactSourceName>
      <ArtifactSource>/solitaire/java/classes</ArtifactSource>
      <ArtifactDestinationName>
        solitaire.class
      </ArtifactDestinationName>
      <ArtifactDestination>classes</ArtifactDestination>
      <ArtifactEntryPoint Value="true"></ArtifactEntryPoint>
      <ArtifactMutable Value="false"></ArtifactMutable>
      <ArtifactPermission></ArtifactPermission>
    </Artifact>
    <Artifact>
      <Guard>($WinHelp$ == true)</Guard>
      <ArtifactSignature>
        4c429e59974e93a3dd6088b7a6eed121
      </ArtifactSignature>
      <ArtifactType>DOCUMENTATION</ArtifactType>
      <ArtifactSourceName>solitaire.hlp</ArtifactSourceName>
      <ArtifactSource>/solitaire/doc</ArtifactSource>
      <ArtifactDestinationName>
        solitaire.hlp
      </ArtifactDestinationName>
      <ArtifactDestination>doc</ArtifactDestination>
      <ArtifactEntryPoint Value="false"></ArtifactEntryPoint>
      <ArtifactMutable Value="false"></ArtifactMutable>
      <ArtifactPermission></ArtifactPermission>
    </Artifact>
    <Artifact>
      <Guard>($HtmlHelp$ == true)</Guard>
      <ArtifactSignature>
        98c975d26650c6bb91f346caac7a3c63
      </ArtifactSignature>
      <ArtifactType>DOCUMENTATION</ArtifactType>
      <ArtifactSourceName>solitaire.html</ArtifactSourceName>
      <ArtifactSource>/solitaire/doc</ArtifactSource>
      <ArtifactDestinationName>
        solitaire.html
      </ArtifactDestinationName>
      <ArtifactDestination>doc</ArtifactDestination>
      <ArtifactEntryPoint Value="false"></ArtifactEntryPoint>
      <ArtifactMutable Value="false"></ArtifactMutable>
      <ArtifactPermission></ArtifactPermission>
    </Artifact>
  </Artifacts>
  <Interfaces>
     <Guard></Guard>
    <Interface>
      <Guard></Guard>
      <InterfaceName>Update</InterfaceName>
      <InterfaceDescription>Update interface</InterfaceDescription>
      <ReleaseDock>heavy.cs.colorado.edu</ReleaseDock>
      <Agent>Update</Agent>
    </Interface>
    <Interface>
      <Guard></Guard>
      <InterfaceName>Reconfigure</InterfaceName>
      <InterfaceDescription>
        Reconfigureinterface
      </InterfaceDescription>
      <ReleaseDock>heavy.cs.colorado.edu</ReleaseDock>
      <Agent>Reconfigure</Agent>
    </Interface>
```

```
    <Interface>
      <Guard></Guard>
      <InterfaceName>Adapt</InterfaceName>
      <InterfaceDescription>Adapt interface</InterfaceDescription>
      <ReleaseDock>heavy.cs.colorado.edu</ReleaseDock>
      <Agent>Adapt</Agent>
    </Interface>
    <Interface>
      <Guard></Guard>
      <InterfaceName>Remove</InterfaceName>
      <InterfaceDescription>Remove interface</InterfaceDescription>
      <ReleaseDock>heavy.cs.colorado.edu</ReleaseDock>
      <Agent>Remove</Agent>
    </Interface>
  </Interfaces>
  <Notifications>
    <Guard></Guard>
    <Notification>
      <Guard></Guard>
      <NotificationName>Winner</NotificationName>
      <NotificationDescription>
        Signals that someone has won the game.
      </NotificationDescription>
    </Notification>
  </Notifications>
  <Services>
     <Guard></Guard>
  </Services>
  <Activities>
     <Guard></Guard>
  </Activities>
</Family>
```

Appendix C

Open Software Description Example

This example is taken directly from the OSD specification [25]. It describes an imaginary implementation of a Solitaire software system. The specification describes two variants of the software system, one for the Microsoft Windows platform and one for the Java platform. The Java implementation has a dependency on a subsystem.

```
<SOFTPKG NAME="com.foobar.www.Solitaire" VERSION="1,0,0,0">
   <TITLE>Solitaire</TITLE>
   <ABSTRACT>Solitaire by FooBar Corporation</ABSTRACT>
   <LICENSE HREF="http://www.foobar.com/solitaire/license.html" />
   <!--FooBar Solitaire is implemented in native code for Win32,
       Java code for other platforms -->
   <IMPLEMENTATION>
      <OS VALUE="WinNT"><OSVERSION VALUE="4,0,0,0"/></OS>
      <OS VALUE="Win95"/>
      <PROCESSOR VALUE="x86" />
      <LANGUAGE VALUE="en" />
      <CODEBASE HREF="http://www.foobar.org/solitaire.cab" />
   </IMPLEMENTATION>
   <IMPLEMENTATION>
      <IMPLTYPE VALUE="Java" />
      <CODEBASE HREF="http://www.foobar.org/solitaire.jar" />
      <!-- The Java implementation needs the DeckOfCards object -->
      <DEPENDENCY>
         <CODEBASE HREF="http://www.foobar.org/cards.osd" />
      </DEPENDENCY>
   </IMPLEMENTATION>
</SOFTPKG>
```

Appendix D

Management Information Format Example

This MIF example tries to mimic the OSD example of Appendix C where possible. Due to the verbosity of MIF many details have been left out of the example. The "..." symbol denotes areas where details have been omitted. In the case of attributes, most details have been omitted, such as the identification number, description when unnecessary, access rights, storage type, and value type. In addition to these omissions, other attribute groups have been omitted entirely since there was no direct relevance to the example; these include the software signature, equivalence, verification, sub-components, superceded components, installation log file, and support.

```
START COMPONENT
    Name = "ComponentID"
    Class = "DMTF|ComponentID|001"
    START ATTRIBUTE
        Name = "Manufacturer"
        ID = 1
        Description = "Manufacturer of this component"
        Access Read-Only
        Storage = Common
        Type = String(64)
        Value = "FooBar Corporation"
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Product"
        ID = 2
        Description = "Product name of this component"
        Access Read-Only
        Storage = Common
        Type = String(64)
        Value = "Solitaire"
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Version"
        ...
        Value = "1.0"
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Serial Number"
        ...
        Value = "1234567890"
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Installation"
        Description = "Automatically assigned date"
        ...
        Value = " "
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Verify"
        Description = "Level of verification"
        Type = START ENUM
            0 = "An error occurred; check status code"
            1 = "This component does not exist"
            2 = "The verify is not supported"
            3 = "Reserved"
            4 = "This component exists, but the"
                "functionality is untested"
            5 = "This component exists, but the"
                "functionality is unknown"
            6 = "This component exists, and is not"
                "functioning correctly"
            7 = "This component exists, and is functioning"
                "correctly"
        END ENUM
        ...
        Value = 2
    END ATTRIBUTE
END GROUP
BEGIN GROUP
    Name = "Software Component Information"
    Class = "DMTF|Software Component Information|002"
    Description = "This group contains additional"
                  "identifying information"
    ...
    START ATTRIBUTE
        Name = "Target Operating System"
        ...
        Type = START ENUM
            0 = "Other"
            1 = "DOS"
            2 = "MACOS"
            3 = "OS2"
            4 = "UNIX"
            5 = "WIN16"
            6 = "WIN32"
            7 = "OPENVMS"
            8 = "NetWare"
        END ENUM
        Value = 6
    END ATTRIBUTE
    START ATTRIBUTE
```

```
        Name = "Language Edition"
        ...
        Value = "en"
    END ATTRIBUTE
    ...
END GROUP
START GROUP
    Name = "Software Location"
    Class = "DMTF|Software Location|001"
    Description = "This group identifies the various"
                  "locations where parts of the component"
                  "have been installed"
    START ATTRIBUTE
        Name = "Index"
        ...
        Type = Integer
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Type"
        ...
        Type = START ENUM
            0 = "Unknown"
            1 = "Other"
            2 = "Product base directory"
            3 = "Product executables directory"
            4 = "Product library directory"
            5 = "Product configuration directory"
            6 = "Product include directory"
            7 = "Product working directory"
            8 = "Product log directory"
            9 = "Shared base directory"
            10 = "Shared executables directory"
            11 = "Shared library directory"
            12 = "Shared include directory"
            13 = "System base directory"
            14 = "System executables directory"
            15 = "System library directory"
            16 = "System configuration directory"
            17 = "System include directory"
            18 = "System log directory"
        END ENUM
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Path"
        ...
    END ATTRIBUTE
END GROUP
START TABLE
    Name = "Software Location"
    Class = "DMTF|Software Location|001"
    { 1, 2, "C:\\SOLITAIRE" }
    { 2, 3, "C:\\SOLITAIRE\\BIN" }
    { 3, 4, "C:\\SOLITAIRE\\DLL" }
END TABLE
START GROUP
    Name = "Component Dependencies"
    Class = "DMTF|Component Dependencies|001"
    Description = "This group identifies dependencies this"
                  "component has on other components"
    START ATTRIBUTE
        Name = "Manufacturer"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Product"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Minimum Version"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Maximum Version"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Type"
        ...
        Type = START ENUM
            0 = "Exclude"
            1 = "Required"
            2 = "One of a set required"
        END ENUM
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Set Index"
        ...
    END ATTRIBUTE
END GROUP
START TABLE
    Name = "Component Dependencies"
    Class = "DMTF|Component Dependencies|001"
    { "Foobar Corporation", "Cards", "2.1", "3.0", 1, 0 }
END TABLE
START GROUP
    Name = "Attribute Dependencies"
    Class = "DMTF|Attribute Dependencies|001"
    Description = "This group identifies dependencies on"
                  "attribute values"
    START ATTRIBUTE
        Name = "Class"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "ID"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Minimum Value"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Maximum Value"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Type"
        ...
        Type = START ENUM
            0 = "Exclude"
            1 = "Required"
            2 = "One of a set required"
        END ENUM
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Set Index"
        ...
    END ATTRIBUTE
END GROUP
START TABLE
    Name = "Attribute Dependencies"
    Class = "DMTF|Attribute Dependencies|001"
    { "DMTF|Video|001", 6, "VGA", "VGA", 2, 1 }
    { "DMTF|Video|001", 6, "SVGA", "SVGA", 2, 1 }
END TABLE
BEGIN GROUP
    Name = "File List"
    Class = "DMTF|File List|001"
    Description = "This group describes the files included"
                  "in the software product"
    START ATTRIBUTE
        Name = "Location"
        Description = "Index into the location table"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Name"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Description"
        ...
    END ATTRIBUTE
END GROUP
START TABLE
    Name = "File List"
    Class = "DMTF|File List|001"
    { 3, "solitaire.exe",
      "Primary executable for this application" }
    { 3, "install.exe",
      "Configuration executable for this application" }
    { 4, "solitaire.dll",
      "Library routines for this application" }
    { 2, "readme.txt", "Readme notes for this application" }
END TABLE
START GROUP
    Name = "Installation"
    Class = "DMTF|Installation|001"
    Description = "This group contains information to allow"
                  "install, update, and deinstall of this"
                  "software component."
    START ATTRIBUTE
        Name = "Type"
        ...
        Type = START ENUM
            0 = "Other"
            1 = "Install"
            2 = "De-install"
        END ENUM
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Name"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Description"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Install size"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Command Line"
        ...
    END ATTRIBUTE
    START ATTRIBUTE
        Name = "Location"
        ...
    END ATTRIBUTE
END GROUP
START TABLE
    Name = "Installation"
    Class = "DMTF|Installation|001"
    { 1, "Deinstall", "Uninstalls this product",
      359.8, "install.exe /d", 5 }
END TABLE
```

Appendix E

Software Dock/InstallShield Comparison Experiments

In order to provide some degree of confidence that an approach similar to the Software Dock framework is feasible, a series of experiments were devised. These experiments compared the deployment routines of a typical software system to the deployment routines of the Software Dock prototype for that same software system. The Java Development Kit (JDK) was selected as the typical software system due to its popularity and relative simplicity. The Microsoft Windows 95 platform was chosen as the platform of comparison since it is ubiquitous and because the JDK is distributed as a self-extracting InstallShield executable; self-extracting InstallShield executables are very prominent on the Microsoft Windows 95 platform.

The experiments test the most common deployment processes, such as install, reconfigure, update, and remove. Time-to-completion is used as the criterion to evaluate each process. Two different tables below (see Table 2 and Table 3) show the detailed timing information for each experiment; a summary table and anecdotal information are presented in Sections 9.4.3.1 and 9.4.3.2.

The install experiment installs the standard configuration of the JDK and includes the retrieval of the install archive itself. The remove experiment removes the standard configuration of the JDK. There are two types of reconfigure experiments, one where software components are removed and one where software components are added. The reconfigure experiments for InstallShield provide results that exclude and include archive transfer time,

Table 2: InstallShield deployment for the Java Development Kit

	Attempt 1	Attempt 2	Attempt 3	Average
JDK 1.1.6 Install				
Transfer	15s	14s	16s	15.0s
Preparation	45s	45s	43s	44.3s
Extract	108s	109s	109s	108.7s
			Total time for JDK 1.1.6 install:	168.0s
JDK 1.1.6 Remove				
	Attempt 1	**Attempt 2**	**Attempt 3**	**Average**
Delete	80s	79s	81s	80.0s
			Total time for JDK 1.1.6 remove:	80.0s
JDK 1.1.6 Reconfigure Remove				
	Attempt 1	**Attempt 2**	**Attempt 3**	**Average**
Preparation	42s	43s	42s	42.3s
Extract	49s	47s	47s	47.7s
			Total time for JDK 1.1.6 reconfigure remove:	90.0s
			Total time for JDK 1.1.6 reconfigure with transfer:	105.0s
JDK 1.1.6 Reconfigure Add				
	Attempt 1	**Attempt 2**	**Attempt 3**	**Average**
Preparation	44s	42s	43s	43.0s
Extract	241s	242s	241s	241.3s
			Total time for JDK 1.1.6 reconfigure add:	284.3s
			Total time for JDK 1.1.6 reconfigure with transfer:	299.3s
JDK 1.1.7 Update				
	Attempt 1	**Attempt 2**	**Attempt 3**	**Average**
Transfer	13s	14s	13s	13.3s
Preparation	43s	42s	42s	42.3s
Extract	96s	96s	90s	94.0s
			Total time for JDK 1.1.7 update:	149.6s

since it is possible to store the archive locally; this is not possible with the Software Dock. The update experiment installs a newer version of the JDK. The Software Dock prototype is able to perform an additional experiment via its adapt process. The adapt experiment verifies that the installed JDK matches its semantic description.

Each experiment was performed at least three times. In carrying out the experiments with the Software Dock prototype, a large degree of variability was encountered. In an effort to account for this, any experiment that exhibited a degree of variability was performed six times. A hypothesis was formed that the combination of Microsoft Windows 95 and the Java Virtual Machine were the cause of the variability. A separate experiment was performed with the Software Dock prototype. This separate experiment performed the install process on a computer running the Solaris operating system. The install process was performed five times and the variability was much lower than on the Microsoft Windows 95 platform (see Table 4).

Each experiment was broken into functional components in order to determine how each deployment process spent its time. Most functional components are self-explanatory, but two in particular require explanation. The “preparation” component is the measure of time from the user invocation of a deployment process until the deployment process displays an interactive interface to the user. For example, it is the time from when the user executes the InstallShield self-extracting archive until InstallShield displays its interactive installation interface. The “overhead” component is only associated with the Software Dock experiments. Both the InstallShield and the Software Dock experiments were based on observed times, but the Software Dock also reported more precise timing information in its log files for some functional components, such as package creation, archive transfer, and archive extraction. The sum of the reported functional component times did not exactly correspond with the observed time of the entire process; the difference between the sum of the functional components and the observed time is recorded as overhead. Each functional component is not always applicable for all experiments and thus is appropriately omitted.

Table 3: Software Dock deployment for the Java Development Kit

JDK 1.1.6 Install							
	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Attempt 5	Attempt 6	Average
Preparation	23s	34s	22s	28s	22s	30s	26.5s
Package	53s	51s	50s	49s	63s	54s	53.3s
Transfer	9s	9s	12s	8s	9s	8s	9.2s
Extract	73s	83s	99s	71s	67s	76s	78.2s
Overhead	3s	15s	2s	2s	5s	2	4.8s
						Total time for JDK 1.1.6 install:	172.0s

JDK 1.1.6 Remove				
	Attempt 1	Attempt 2	Attempt 3	Average
Delete	34s	38s	38s	36.7s
			Total time for JDK 1.1.6 remove:	36.7s

JDK 1.1.6 Reconfigure Remove				
	Attempt 1	Attempt 2	Attempt 3	Average
Preparation	4s	4s	4s	4.0s
Delete	36s	38s	35s	36.3s
			Total time for JDK 1.1.6 reconfigure remove:	40.3s

JDK 1.1.6 Reconfigure Add							
	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Attempt 5	Attempt 6	Average
Preparation	5s	4s	3s	3s	4s	4s	3.8s
Package	34s	39s	22s	22s	46s	24s	31.2s
Transfer	3s	3s	3s	3s	3s	3s	3.0s
Extract	42s	83s	72s	78s	58s	63s	66.0s
Overhead	29s	2s	1s	21s	2s	1	9.3s
						Total time for JDK 1.1.6 reconfigure add:	113.3s

JDK 1.1.6 Adapt				
	Attempt 1	Attempt 2	Attempt 3	Average
Verify	28s	26s	27s	27.0s
			Total time for JDK 1.1.6 adapt:	27.0s

JDK 1.1.7 Update							
	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Attempt 5	Attempt 6	Average
Preparation	36s	36s	33s	30s	30s	34s	33.2s
Delete	34s	35s	35s	33s	39s	45s	36.8s
Package	48s	44s	44s	45s	44s	44s	44.8s
Transfer	8s	9s	8s	9s	7s	7s	8.0s
Extract	62s	60s	67s	62s	57s	54s	60.3s
Overhead	11s	2s	5s	2s	3s	2	4.2s
						Total time for JDK 1.1.7 update:	187.3s

Table 4: Software Dock deployment for the Java Development Kit on Solaris

JDK 1.1.6 Install						
	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Attempt 5	Average
Package	65s	66s	62s	65s	68s	65.2s
Transfer	3s	2s	2s	3s	2s	2.4s
Extract	37s	42s	37s	41s	42s	39.8s
Overhead	1s	1s	2s	1s	1s	1.2s
					Total time for JDK 1.1.6 install:	108.6s