

Specifying the Deployable Software Description Format in XML

CU-SERL-207-99

March 31, 1999

Richard S. Hall, Dennis Heimbigner, Alexander L. Wolf
Software Engineering Research Laboratory
University of Colorado at Boulder
Boulder, Colorado 80309-0430 USA
[rickhall,dennis,alw]@cs.colorado.edu

Abstract. *Large network environments, such as enterprise Intranets and the Internet, have moved to the center stage in the field of software deployment. The connectivity inherent in these environments provides a new tool that deployment solutions can leverage to perform a comprehensive software deployment life cycle. This life cycle covers a wide range of deployment tasks, including release, install, update, reconfigure, adapt, and remove. But network solutions place a premium on task automation, which requires complete, concise descriptions of the deployment requirements for software systems. The deployment information to be captured for a software system includes its dependencies, constraints, and alternative configurations. This paper presents the Deployable Software Description (DSD) format, a standard schema for describing deployment requirements for the purpose of automating the tasks of the software deployment life cycle. The DSD format is described and compared to other technologies addressing many of the same concerns.*

1 Introduction

Large network environments, such as enterprise Intranets and the Internet, have moved to the center stage in the field of software deployment. The connectivity provided by these environments provides a new tool that deployment solutions can leverage to perform a comprehensive software deployment life cycle [8]. This life cycle covers a wide range of deployment tasks, including release, install, update, reconfigure, adapt, and remove. But network solutions place a premium on task automation, a goal that is not easily attained with the current lack of standardized support for software deployment.

In order to support the automation of software deployment processes, it is necessary to introduce specific enabling technologies. One important technology is software system and component description languages and schema. These languages and schema are used to provide declarative information necessary to automate or simplify most software deployment tasks. Without a description of a software system's constraints, dependencies, and configuration it is nearly impossible to perform any software deployment task.

To address some of these issues, a handful of software system and component description languages and schema have been introduced to aid the software producer in providing high-level deployment services. Specific efforts are attempting to create standard syntax and semantics for describing software systems in order to facilitate deployment and related activities. These efforts include the Open Software Description (OSD) [11], the Management Information Format (MIF) [5], the Application Management Specification [17], and the Defense Information Infrastructure Common Operating Environment (DII COE) [12].

The purpose of this paper is to discuss a related schema used in the Software Dock [8] research project, which is creating a unified framework for supporting software deployment. A critical piece to the Software Dock is the definition of a standard schema to describe software systems for deployment, namely the Deployable Software Description (DSD) format. This paper introduces a definition of a software system in Section 2 and discusses specific issues regarding software system description in Section 3. The DSD format is presented in Section 4, followed by a discussion of how a DSD specification is processed in Section 5. Related work is presented in Section 6, followed by the conclusions.

2 Definition of a Software System

When considering the task of describing a software system for deployment, 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 of a software system is nothing more than the collection of artifacts that comprise a given software system, where an artifact is anything that may be contained in a file.

This initial definition is clearly only sufficient for the most basic instances of a software system. Invariably the artifacts that comprise a software system vary according to some set of relationships. For example, the collection of artifacts that comprise a software system usually varies with respect to the properties of platform, revision, or configuration. Some properties denote internal relationships, such as revisions or configurations. Other properties denote external 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. Using the properties it is possible to select the specific artifacts to deploy to a given consumer site based on the property values.

Further, it is also necessary to capture subsequent changes to the state of the target consumer site. 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.

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

3 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 described in [8] 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. In order to provide support for software deployment, a language or schema must capture enough information to support these deployment processes.

Large network environments further complicate support for the software deployment life cycle. In such environments, variability is normal and uncontrollable. Other 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 networked environments because of the inherent variability such as platform, platform configuration, software configuration, and software revision.

The last form of variability results from software system revision histories. 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. 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. 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.

4 The Deployable Software Description (DSD) Format

The Deployable Software Description (DSD) format described in this section is an application of the Extensible Markup Language (XML) [2]. DSD is a vocabulary for describing software systems and their complex internal and external dependencies and relationships for heterogeneous software consumers. It 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.

DSD was created to meet the requirements of software system description for deployment as discussed in [10]. In brief, the software system description requirements first state that it is necessary to have an implicit consumer site model that describes the properties, constraints, and resources of the target deployment site because it is not possible to fully deploy a software system in the absence of consumer site information. Second, the software system description itself must capture information about the configuration, assertions, dependencies, artifacts, and specialized activities for a specific software system.

4.1 Approach

DSD uses a standard schema approach to describe software systems for deployment. This approach models software systems and computing sites as nested collections of properties. The decision to use a schema approach requires selecting what to describe. In the simplest case, the schema may describe a single software system configuration: a single revision and variant. In the most complex case, a software system schema may describe an entire software family: all revisions and variants. These two extremes are the endpoints of a spectrum in which various combinations and hybrids may occur.

The specific approach taken in DSD is to describe an entire software family within a single schema description. 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 extended as a direct result of component-based software development. Maintaining a range of revisions in a single description simplifies revision resolution and access.

Second, the complete family description approach provides flexibility. With this approach it is possible to describe a dependency with respect to a specific configuration of a software system, rather than with respect to 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.

Finally, specifying a complete family increases reuse between the revisions in the family. Significant portions of a system may be 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 DSD specifications to include references to file artifact lists could also 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.

DSD also makes the assumption that a consumer site description is accessible. The consumer site description must model 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 potential. Other approaches tend to include a limited number of standard consumer site properties or they merely ignore the issue all together.

An expression language is also an implicit part of the DSD. The expression language is not specified by DSD. An arbitrary expression language is appropriate as long as it is extended to allow expressions over the software system properties and the consumer site properties. For example, if the DSD specification for a software family has an “online 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 main purpose of the expression language is to place guards strategically throughout a given DSD specification to create a mapping from the software system properties to the schema elements themselves. A guard is simply a

```

<!ELEMENT Family (Id, ExternalProperties, Properties, Composition,
    Assertions, Dependencies, Artifacts, Notifications, Interfaces,
    Services, Activities)>
<!ELEMENT VarType EMPTY>
<!ATTLIST VarType Value (string | boolean | double) #REQUIRED>
<!ELEMENT Id (Name, Description, Producer, (License)?, Logo,
    Signature)>
<!ELEMENT ExternalProperties (ExternalProperty)*>
<!ELEMENT ExternalProperty (Name, VarType, Description, Value)>
<!ELEMENT Properties (Property)*>
<!ELEMENT Property (Name, VarType, Description, DefaultValue,
    DefaultEnabled, DefaultDisabled, TopLevel, Values)>
<!ELEMENT Composition (CompositionRule)*>
<!ELEMENT CompositionRule (Condition, ControlProperty, Relation,
    RuleProperties)>
<!ELEMENT Relation EMPTY>
<!ATTLIST Relation Value (anyof | oneof | excludes | includes)
    #REQUIRED>
<!ELEMENT Assertions (Guard, (Assertions | Assertion)*)>
<!ELEMENT Assertion (Guard, Condition, Description)>
<!ELEMENT Dependencies (Guard, (Dependencies | Dependency)*)>
<!ELEMENT Dependency (Guard, Condition, Description, Resolution,
    Constraints)>
<!ELEMENT Artifacts (Guard, (Artifacts | Artifact)*)>
<!ELEMENT Artifact (Guard, Signature, ArtifactType, SourceName,
    Source, DestinationName, Destination, EntryPoint, Mutable,
    Permission, DiskFootPrint)>
<!ELEMENT Notifications (Guard, (Notifications | Notification)*)>
<!ELEMENT Notification (Guard, Name, Description)>
<!ELEMENT Interfaces (Guard, (Interfaces | Interface)*)>
<!ELEMENT Interface (Guard, Name, Description)>
<!ELEMENT Services (Guard, (Services | Service)*)>
<!ELEMENT Service (Guard, Name, Description)>
<!ELEMENT Activities (Guard, (Activities | Activity)*)>
<!ELEMENT Activity (Guard, Name, Action, When, Description)>

```

Figure 1: Main XML DTD definitions for the DSD schema elements.

schema expression that evaluates to either true or false. The guard expression is arbitrary and can reference both internal software family properties defined in the property section of a DSD specification, and external properties of the target consumer 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. Refer Appendix B for example expressions and usage.

4.2 XML Document Type Definition

The XML document type definition (DTD) for DSD 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 important 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 is necessary to avoid naming conflicts.

The XML DTD definition for the DSD schema elements discussed in the following sections is presented in Figure 1. The following sections discuss each specific schema element and its associated collection. For example, an Artifact schema element has an associated Artifacts collection schema element that may contain zero or more Artifact elements. The name of a collection schema element is merely the plural form of the specific schema element. Also, it is possible for both collections and scalar schema elements to have associated guard expressions; guard expressions

are used to determine applicability with respect to a given set of property values as discussed in Section 4.1. For the sake of brevity, collections and guard expressions will not be discussed further in the following sections.

A simple DSD specification is presented in Appendix B to illustrate the proper usage of the DSD schema elements discussed below. The example DSD specification in Appendix B is broken into sections that parallel the following sections.

4.2.1 Family

The `Family` schema element corresponds to the top-level XML document element. As the type name suggests, `Family` describes a software family. 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`.

4.2.2 Id

The `Id` schema element provides a human-readable description of the system. The first three elements of `Id` are simple character string values that represent the unique name, description, and producer of the software family. The `License` element is an optional reference to a licensing document. The `Logo` element is a reference to a logo image to customize the installation processes, and the final element, `Signature`, 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. The `Id` section of the DSD format is not considered complete, rather it is illustrative of some of the necessary concepts for identification; future versions of the DSD will expand this section.

4.2.3 ExternalProperties

The `ExternalProperties` schema element is a container for storing 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. An `ExternalProperties` schema element contains zero or more `ExternalProperty` elements. An `ExternalProperty` element is basically a name-value pair, where the “name” is the name of the external property and the “value” is the value of the external property at the time of usage; a property type and description are also specified. The external property section is not completed during the description creation process, since external property values are not known in advance. Only information about the external property themselves are completed. The values of the external 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.

4.2.4 Properties

A `Property` describes a specific internal property of a software family. 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 `DefaultEnabled` and `DefaultDisabled` 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, `Values`, 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 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 special 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. Additionally, version numbers often “encode” more than just the time-ordered revisions of a software system. Often they are used to encode additional properties such as functionality. Placing less importance on the version number and providing a means to describe capabilities as properties creates a more flexible and informative software description.

4.2.5 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 internal properties of a software family. 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 is enforced; if the boolean result is “true” then the rule is enforced, if the boolean result is “false” then the rule is not enforced. 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 exactly 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`, respectively. 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 rule 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 the specific property.

The final element, `ControlProperty`, 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 `CompositionRule`, but would have resulted in less flexibility. In the current definition, properties may participate in arbitrary combinations of hierarchical or non-hierarchical relationships.

4.2.6 Assertions and Dependencies

`Assertion` and `Dependency` both describe a constraint of the software system over internal and external property values. An `Assertion` is intended to test a constraint that cannot or should not be resolved in the face of a conflict. On the other hand, a `Dependency` tests a constraint that is resolvable in the face of a conflict.

The `Condition` element of an `Assertion` is 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 assertions are 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.

The `Condition` element of a `Dependency` is a schema expression that 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 re-

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

4.2.7 Artifacts

An `Artifact` describes a specific software system artifact, represented as a file, of the software family. The `Artifact` is the main building block of a software family and any specific software system configuration. The `Signature` 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 `SourceName` element specifies the name of the artifact as it appears in the source for the specified artifact. The `Source` element may reference a directory or an archive that contains the artifact. The `Source` is specified as a direct path or via a URL. The `DestinationName` specifies the name that the artifact must have in its final destination; the destination name and the source name may differ. The `Destination` element specifies the destination directory for the artifact relative to the base directory where the software system is installed; the destination is relative since the installation location varies for each deployment.

The `EntryPoint` 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. `Mutable` 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 `Permission` element 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 universal read-only permission. The last element, `DiskFootPrint`, is the size of the artifact in kilobytes.

4.2.8 Notifications, Interfaces, and Services

The DTD definitions of the `Notification`, `Interface`, and `Service` schema elements are all very similar, but each is used to describe a distinct aspect of a software system. A `Notification` describes a specific notification that is generated by the software system, an `Interface` describes a single-site specific interface provided by the software system, and a `Service` describes a specific global service provided by the software system. These elements are not intended to depict low-level notifications, interfaces, or services provided by the software system, rather they are intended to depict high-level, management-related notions. For example, notifications describe administrative events such as a server failure, while interfaces describe deployment or administrative operations, and services describe enterprise services such as an ODBC source.

Each of these schema elements is identified by a name and may also have a human-readable description. DSD does not prescribe how interfaces are activated, notifications are monitored, or services accessed; these issues are outside the scope of the current DSD definition. Proprietary and future implementations of DSD will most likely extend these definitions as necessary.

4.2.9 Activities

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 `Activity` element has an `Action` element that is a placeholder to describe the action to perform. The `When` element describes when the activity should occur; the contents are not specified by DSD but a reasonable example is "after update." The `De-`

scripti`on` 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.

5 Generic Deployment Process Algorithm

The approach of DSD is to describe a software system in terms of properties. A valid set of software system property values represents a particular valid configuration of a software system. Choosing a particular configuration involves a three-step process. First, property values are chosen subject to the constraints implied by the composition rules. Second, the property values are used to select sets of artifacts subject to assertion and dependency constraints. Finally, the set of selected artifacts is extracted as the desired configuration.

Each of the deployment processes creates a new set of valid property values where in some cases the property value set is empty. The install, update, adapt, reconfigure, and remove software deployment processes can be characterized as the transformation of one software system configuration to another. This transformation is based on the set of property values for a given software system configuration. Given a new set of property values, the deployment process simply transforms its current configuration to the new configuration by performing differential processing over the applicable schema elements of the DSD specification. As a common example, if the version of a software system is changed from “1.0” to “1.1,” then all of the artifacts associated with version “1.0” are removed, the artifacts associated with version “1.1” are added, and any common artifacts are left untouched.

6 Related Work

Configuration management technologies have introduced sophisticated software system modeling languages, such as Adele [7] and PCL [18]. 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 main contribution of Adele is the use of constraints over these attributes to express the composition of valid configurations.

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 definition includes attributes, parts or sub-components, relationships, and physical objects (e.g., files). The family attributes are used to characterize potential variability. Both Adele and PCL are general modeling languages and do not provide any standard schema definitions for software deployment. It is possible, however, to build a deployment schema, such as DSD, using these languages.

The Open Software Description (OSD) [11] format was submitted to the W3 Consortium 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 [15], is related to Microsoft's Channel Definition Format (CDF) [6] for “push” content. The syntax for both OSD and CDF are based on the Extensible Markup Language (XML) [2]. Overall, OSD is too simplistic for deployment process automation; it merely provides a means to bundle multiple archives and their dependencies. For a more detailed evaluation of OSD refer to [9].

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) [3], which creates a common interface layer to access management information on computing systems. An effort related to DMI is the Management Information Format (MIF) [5], 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 [4] created by the Application Management working group.

A superset to the Software MIF was created by Tivoli and is called the Application Management Specification (AMS) [17]. 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 [16], where there is a central administration authority that is responsible for maintaining the state of deployed software systems. For a more detailed evaluation of MIF that is also relevant to AMS refer to [9].

The Defense Information Infrastructure Common Operating Environment (DII COE) [12] is a Department of Defense effort to restrict the set of components used to build their software systems. The DII COE supports, among other things, a standard means for packaging components for delivery and installation. These packages are called segments [13], where each segment is a separate, installable entity. The DII COE segment describes the constraints, dependencies, and artifacts of a software system. High-level software deployment process support is provided in the form of scripts.

With respect to MIF, AMS, and DII COE segments, each suffers from the same significant shortcoming; none support variability. Each assumes that a central administration authority dictates a standard configuration of a software system. In general, these description technologies are intended for enterprise administration, not for managing deployment in general. In addition, none are specifically intended for Internet distribution or have support for general software release; for example, these languages do not even specify a source for their file artifacts.

The Redhat Package Manager (RPM) [1] is a tool for the Linux user community that provides many software deployment features. RPM is also an 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. Several support tools for retrieving and installing software packages from a network accompany RPM. RPM does not assume a central authority and is actually intended for individual consumer sites. RPM does not extend easily to groups of managed consumer sites. RPM only provides limited configuration selection. 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.

7 Conclusion

Large networked environments are playing an increasingly important role in how software deployment is performed. With the connectivity provided by networks, demand is being created for deployment process automation. New technologies in the area of software system description for deployment are necessary to meet this demand.

The DSD format presented in this paper is our initial attempt to create a standard mechanism for describing software systems for deployment. DSD describes the basic semantic elements of a software system that are necessary for nearly any software deployment task. DSD differs from other software system description approaches because its descriptions are not static "inventory" descriptions of the deployed software system; rather it enables the dynamic selection of a software system configuration through schema interpretation and manipulation. The end result is that software systems described in DSD can have their deployment processes automated to a large degree by standard process definitions that are parameterized by the DSD description.

The Software Dock research project [8] is creating a deployment framework with an implementation of the DSD format at its core. A prototype of the Software Dock framework exists. Using DSD specifications, the prototype is capable of automating recursive install and differential update, reconfigure, adapt, and remove processes for configurable software systems and content. Work on the Software Dock framework and the DSD format continues to address both enterprise and platform-specific deployment issues.

References

1. E. C. Bailey. "Maximum RPM," Red Hat Software, Inc., ISBN: 1-888172-78-9, Feb. 1997.
2. T. Bray. "Extensible Markup Language (XML): Part I. Syntax". Textuality, Vancouver, BC, Canada. <http://www.w3.org/pub/WWW/TR/WD-xml-lang.html>.
3. Desktop Management Task Force. "Desktop Management Interface Specification," Version 2.0s, June 24, 1998.
4. Desktop Management Task Force, "Software Standard Groups Definition, Version 2.0," Mar. 27, 1996. <http://www.dmtf.org/tech/apps.html>.
5. Desktop Management Task Force, "Enabling your product for manageability with MIF files," Nov. 1994.
6. C. Ellerman. "Channel Definition Format", 1997. Microsoft Corp, Redmond, WA.
7. J. Estublier and R. Casallas. "The Adele Configuration Manager," Configuration Management, Wiley, 1994, pp. 99-134.
8. R. S. Hall, D. Heimbigner, and A. L. Wolf. "A Cooperative Approach to Support Software Deployment Using the Software Dock," to appear in the Proceedings of the 1999 International Conference on Software Engineering, IEEE Computer Society, May 1999.
9. 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.
10. 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.
11. 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>.
12. Joint Interoperability and Engineering Organization. "Defense Information Infrastructure Common Operating Environment Baseline Specifications," 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
13. Joint Interoperability and Engineering Organization. "How to Segment Guide," Version 4.0, Defense Information Systems Agency, Dec. 30 1996. http://spider.osfl.disa.mil/cm/how_to/howtoseg.pdf.
14. Marimba, Inc. "Castanet Product Family," 1998. http://www.marimba.com/datasheets/castanet-3_0-ds.html.
15. Microsoft Corp, Redmond, WA. "Zero Administration Initiative", 1998. <http://www.microsoft.com/windows/innovation>.
16. Tivoli Systems. "Tivoli Enterprise Overview," 1998. http://www.tivoli.com/o_products/html/enterprise36.html.
17. Tivoli Systems. "Application Management Specification," Version 2.0, Nov. 5 1997. http://www.tivoli.com/o_products/html/body_ams_spec.html.
18. 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.

Appendix A Deployable Software Description XML DTD

The DSD specification presented here is an application of the Extensible Markup Language (XML) [2], 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, ExternalProperties, Properties, Composition, Assertions,
  Dependencies, Artifacts, Notifications, Interfaces, Services, Activities)>

<!ELEMENT Guard (#PCDATA)>
<!ELEMENT Condition (#PCDATA)>
<!ELEMENT DiskFootPrint (#PCDATA)>
<!ELEMENT VarType EMPTY>
<!ATTLIST VarType Value (string | boolean | double) #REQUIRED>
<!ELEMENT Value (#PCDATA)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Signature (#PCDATA)>

<!ELEMENT Id (Name, Description, Producer, (License)?, Logo, Signature)>
<!ELEMENT Producer (#PCDATA)>
<!ELEMENT License (#PCDATA)>
<!ELEMENT Logo (#PCDATA)>

<!ELEMENT ExternalProperties (ExternalProperty)*>
<!ELEMENT ExternalProperty (Name, VarType, Description, Value)>

<!ELEMENT Properties (Property)*>
<!ELEMENT Property (Name, VarType, Description, DefaultValue, DefaultEnabled,
  DefaultDisabled, TopLevel, Values)>
<!ELEMENT DefaultValue (#PCDATA)>
<!ELEMENT DefaultEnabled (#PCDATA)>
<!ELEMENT DefaultDisabled (#PCDATA)>
<!ELEMENT TopLevel EMPTY>
<!ATTLIST TopLevel Value (true | false) #REQUIRED>
<!ELEMENT Values (Value)*>

<!ELEMENT Composition (CompositionRule)*>
<!ELEMENT CompositionRule (Condition, ControlProperty, Relation, RuleProperties)>
<!ELEMENT ControlProperty (Name)?>
<!ELEMENT Relation EMPTY>
<!ATTLIST Relation Value (anyof | oneof | excludes | includes ) #REQUIRED>
<!ELEMENT RuleProperties (Name)*>

<!ELEMENT Assertions (Guard, (Assertions | Assertion)*)>
<!ELEMENT Assertion (Guard, Condition, Description)>

<!ELEMENT Dependencies (Guard, (Dependencies | Dependency)*)>
<!ELEMENT Dependency (Guard, Condition, Description, Resolution, Constraints)>
<!ELEMENT Resolution (#PCDATA)>
<!ELEMENT Constraints (ExternalProperty)*>

<!ELEMENT Artifacts (Guard, (Artifacts | Artifact)*)>
<!ELEMENT Artifact (Guard, Signature, ArtifactType, SourceName, Source,
  DestinationName, Destination, EntryPoint, Mutable, Permission, DiskFootPrint)>
<!ELEMENT ArtifactType (#PCDATA)>
```

```
<!ELEMENT SourceName (#PCDATA)>
<!ELEMENT Source (#PCDATA)>
<!ELEMENT DestinationName (#PCDATA)>
<!ELEMENT Destination (#PCDATA)>
<!ELEMENT EntryPoint EMPTY>
<!ATTLIST EntryPoint Value (true | false) #REQUIRED>
<!ELEMENT Mutable EMPTY>
<!ATTLIST Mutable Value (true | false) #REQUIRED>
<!ELEMENT Permission (#PCDATA)>

<!ELEMENT Notifications (Guard, (Notifications | Notification)*)>
<!ELEMENT Notification (Guard, Name, Description)>

<!ELEMENT Interfaces (Guard, (Interfaces | Interface)*)>
<!ELEMENT Interface (Guard, Name, Description)>

<!ELEMENT Services (Guard, (Services | Service)*)>
<!ELEMENT Service (Guard, Name, Description)>

<!ELEMENT Activities (Guard, (Activities | Activity)*)>
<!ELEMENT Activity (Guard, Name, Action, When, Description)>
<!ELEMENT Action (#PCDATA)>
<!ELEMENT When (#PCDATA)>
```

Appendix B Example DSD Specification

This section introduces an example DSD specification. The example follows the lead of the OSD specification [11] and uses a fictitious implementation of Solitaire. The example illustrates most of the main capabilities of the DSD description format. In particular, it illustrates support for multiple revisions and variants of the Solitaire system for Windows 95 and for the Java Virtual Machine. Special notice should be given to the fact that artifact sharing across revisions and variants is explicitly supported; 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. 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. The following sections create a complete DSD specification when combined. The dialect of the DSD specification presented here is from the Software Dock research project, therefore the expression language and some additional attributes are Software Dock specific.

B.1 Family, Id, and ExternalProperties

```
<Family>
  <Id>
    <Name>Solitaire</Name>
    <Description>The game of Solitaire</Description>
    <Producer>FooBar Corporation</Producer>
    <License>http://www.foobar.com/license.html</License>
    <Logo>http://www.foobar.com/solitairelogo.gif</Logo>
    <Signature>f2a3b8c32091fd46a785c39723e289a9</Signature>
  </Id>
  <ExternalProperties>
    <ExternalProperty>
      <Name>OS</Name>
      <VarType Value="string"></VarType>
      <Description>Operating system</Description>
      <Value></Value> <!-- recorded at deployment -->
    </ExternalProperty>
  </ExternalProperties>
```

B.2 Properties

```
<Properties>
  <Property>
    <Name>Implementation</Name>
    <VarType Value="string"></VarType>
    <Description>
      Selects implementation
    </Description>
    <DefaultValue>Java</DefaultValue>
    <DefaultEnabled>Native</DefaultEnabled>
    <DefaultDisabled>Java</DefaultDisabled>
    <TopLevel Value="true"></TopLevel>
    <Values>
      <Value>Native</Value>
      <Value>Java</Value>
    </Values>
  </Property>
  <Property>
    <Name>Online Help</Name>
    <VarType Value="boolean"></VarType>
    <Description>
      Includes online help documentation
    </Description>
    <DefaultValue>>true</DefaultValue>
    <DefaultEnabled></DefaultEnabled>
```

```

    <DefaultDisabled></DefaultDisabled>
    <TopLevel Value="true"></TopLevel>
    <Values></Values>
  </Property>
  <Property>
    <Name>WinHelp</Name>
    <VarType Value="boolean"></VarType>
    <Description>
      Includes WinHelp documentation
    </Description>
    <DefaultValue>false</DefaultValue>
    <DefaultEnabled></DefaultEnabled>
    <DefaultDisabled></DefaultDisabled>
    <TopLevel Value="false"></TopLevel>
    <Values></Values>
  </Property>
  <Property>
    <Name>HtmlHelp</Name>
    <VarType Value="boolean"></VarType>
    <Description>
      Includes HTML documentation
    </Description>
    <DefaultValue>true</DefaultValue>
    <DefaultEnabled></DefaultEnabled>
    <DefaultDisabled></DefaultDisabled>
    <TopLevel Value="false"></TopLevel>
    <Values></Values>
  </Property>
</Properties>

```

B.3 Composition

```

<Composition>
  <!-- This rule forces the "Implementation" property to be Java if
        the operating system is not a Windows variant -->
  <CompositionRule>
    <Condition>
      (($OS$ != "Win95") AND ($OS$ != "WinNT"))
    </Condition>
    <ControlProperty></ControlProperty>
    <Relation Value="excludes"></Relation>
    <RuleProperties>
      <Name>Implementation</Name>
    </RuleProperties>
  </CompositionRule>
  <!-- This rule indicates that if the "Online Help" propertied is
        selected, then either the "WinHelp" or "HtmlHelp" must be
        selected -->
  <CompositionRule>
    <Condition>($Online Help$ == true)</Condition>
    <ControlProperty>
      <Name>Online Help</Name>
    </ControlProperty>
    <Relation Value="oneof"></Relation>
    <RuleProperties>
      <Name>WinHelp</Name>
      <Name>HtmlHelp</Name>
    </RuleProperties>
  </CompositionRule>
  <!-- This rule forces the "WinHelp" property to be false if the
        operating system is not a Windows variant -->
  <CompositionRule>
    <Condition>

```

```

    (($OS$ != "Win95") AND ($OS$ != "WinNT"))
  </Condition>
</ControlProperty></ControlProperty>
<Relation Value="excludes"></Relation>
<RuleProperties>
  <Name>WinHelp</Name>
</RuleProperties>
</CompositionRule>
</Composition>

```

B.4 Assertions and Dependencies

```

<Assertions>
  <Guard></Guard>
  <Assertion>
    <Guard></Guard>
    <Condition>
      (($OS$=="Win95") || ($OS$=="WinNT") ||
      ($OS$=="Solaris") || ($OS$=="Linux"))
    </Condition>
    <Description>
      Only supported on Win95, WinNT, Solaris, and Linux.
    </Description>
  </Assertion>
</Assertions>
<Dependencies>
  <Guard></Guard>
  <Dependency>
    <Guard>($Implementation$ == "Java")</Guard>
    <Condition>
      (!installed("Cards"))
    </Condition>
    <Description>
      Java version depends on deck of cards component.
    </Description>
    <Resolution>Install</Resolution>
    <Constraints></Constraints>
    <!-- Software Dock specific attributes -->
    <ReleaseDock>www.cards.com</ReleaseDock>
    <Family>Cards</Family>
  </Dependency>
</Dependencies>

```

B.5 Artifacts

```

<Artifacts>
  <Guard></Guard>
  <Artifact>
    <Guard>($Implementation$ == "Native")</Guard>
    <Signature>
      5d929a2486c67e71f9231697119452d0
    </Signature>
    <ArtifactType>EXECUTABLE</ArtifactType>
    <SourceName>solitaire.exe</SourceName>
    <Source>/solitaire/win/bin</Source>
    <DestinationName>
      solitaire.exe
    </DestinationName>
    <Destination>bin</Destination>
    <EntryPoint Value="true"></EntryPoint>
    <Mutable Value="false"></Mutable>
    <Permission></Permission>
    <DiskFootPrint>485430.0</DiskFootPrint>
  </Artifact>
</Artifacts>

```

```

</Artifact>
<Artifact>
  <Guard>($Implementation$ == "Java")</Guard>
  <Signature>
    96429c4a616df82513e45050ac03d55e
  </Signature>
  <ArtifactType>CLASSFILE</ArtifactType>
  <SourceName>solitaire.class</SourceName>
  <Source>/solitaire/java/classes</Source>
  <DestinationName>
    solitaire.class
  </DestinationName>
  <Destination>classes</Destination>
  <EntryPoint Value="true"></EntryPoint>
  <Mutable Value="false"></Mutable>
  <Permission></Permission>
  <DiskFootPrint>128576.0</DiskFootPrint>
</Artifact>
<Artifact>
  <Guard>($WinHelp$ == true)</Guard>
  <Signature>
    4c429e59974e93a3dd6088b7a6eed121
  </Signature>
  <ArtifactType>DOCUMENTATION</ArtifactType>
  <SourceName>solitaire.hlp</SourceName>
  <Source>/solitaire/doc</Source>
  <DestinationName>
    solitaire.hlp
  </DestinationName>
  <Destination>doc</Destination>
  <EntryPoint Value="false"></EntryPoint>
  <Mutable Value="false"></Mutable>
  <Permission></Permission>
  <DiskFootPrint>12202.0</DiskFootPrint>
</Artifact>
<Artifact>
  <Guard>($HtmlHelp$ == true)</Guard>
  <Signature>
    98c975d26650c6bb91f346caac7a3c63
  </Signature>
  <ArtifactType>DOCUMENTATION</ArtifactType>
  <SourceName>solitaire.html</SourceName>
  <Source>/solitaire/doc</Source>
  <DestinationName>
    solitaire.html
  </DestinationName>
  <Destination>doc</Destination>
  <EntryPoint Value="false"></EntryPoint>
  <Mutable Value="false"></Mutable>
  <Permission></Permission>
  <DiskFootPrint>10908.0</DiskFootPrint>
</Artifact>
</Artifacts>

```

B.6 Notifications, Interfaces, and Services

```

<Notifications>
  <Guard></Guard>
  <Notification>
    <Guard></Guard>
    <Name>Winner</Name>
    <Description>
      Signals that someone has won the game.
    </Description>
  </Notification>
</Notifications>

```

```

    </Description>
  </Notification>
</Notifications>
<Interfaces>
  <Guard></Guard>
  <Interface>
    <Guard></Guard>
    <Name>Update</Name>
    <Description>Update interface</Description>
  </Interface>
  <Interface>
    <Guard></Guard>
    <Name>Reconfigure</Name>
    <Description>Reconfig interface</Description>
  </Interface>
  <Interface>
    <Guard></Guard>
    <Name>Adapt</Name>
    <Description>Adapt interface</Description>
  </Interface>
  <Interface>
    <Guard></Guard>
    <Name>Remove</Name>
    <Description>Remove interface</Description>
  </Interface>
</Interfaces>
<Services>
  <Guard></Guard>
</Services>

```

B.7 Activities

```

<Activities>
  <Guard></Guard>
</Activities>
</Family>

```