Requirements for Software Deployment Languages and Schema

Richard S. Hall, Dennis Heimbigner, Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
{rickhall, dennis, alw}@cs.colorado.edu

Abstract. Software distribution via networks provides timeliness and continual evolution that is not possible with physical media distribution methods. Organizations such as Microsoft, Marimba, and the Desktop Management Task Force are strengthening their efforts to package software systems in ways that are conducive to network distribution. The result of these efforts has led to the creation of software description languages and schema, but they do not address deployment issues in a complete, systematic fashion. The contribution of this paper is to define and explore the requirements for software deployment languages and schema.

1 Introduction

Software distribution methods are evolving constantly. The once preferred floppy disk has given way to the CD-ROM, which is now being challenged by wide-area networks as a software distribution mechanism. Current bandwidth considerations limit the volume of data that can be distributed over a network connection, but it is clear that software distribution via a network connection is growing in importance.

Given benefits such as timeliness, continual evolution, and market access, it is evident that technologies to support network software distribution are important and must be created. One area in particular that is starting to see activity is software deployment languages and schema. Efforts such as the Software Dock [3], the Desktop Management Task Force (DMTF) [1], and the Open Software Description (OSD) [5] are trying to create standard syntax and schema for describing software systems in order to facilitate deployment over networks. These technologies share a common approach with other work, such PCL [7], Adele [2], DCDL [6], in that they are trying to define a standard terminology that can be used to reason about software for specific tasks.

The contribution of this paper is to define and explore what is required in a software description language or schema in order to support software deployment. Additional details and an evaluation of OSD and MIF can be found in [4].

B. Magnusson (Ed.): ECOOP 98, SCM-8, LNCS 1439, pp. 198-203, 1998

2 Software Deployment Language and Schema Requirements

Software deployment is a collection of interrelated activities, referred to as the software deployment life cycle, 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, adapt, activate, de-activate, de-install, and de-release. 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 semantic knowledge that is rich and rigorous enough to support the automation of the software deployment life cycle. Given such semantic descriptions, generic solutions to software deployment tasks are possible by combining software product knowledge with consumer site knowledge.

In order to provide such semantic descriptions of software systems and consumer sites it is necessary to adopt a semantic model. A common approach, and one that is assumed in this paper, is that software systems and computing sites are objects that can be modeled as a collection of attributes or 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 used throughout this paper and, in general, is used by both OSD and MIF.

2.1 Consumer Site Description

There are two classes of participants in the software deployment problem space, namely producers and consumers. The purpose of the consumer description is to provide a context about the site into which a software system is to be situated; this is essential to fully describe a software system or component. The consumer site description and the software system description, should be viewed as two halves to a whole, rather than two distinct entities.

In the end, the consumer site description is combined with the software system description to determine how deployment processes should be performed. The result of such deployment activities is a modification to the consumer site description in order to record the results of the deployment activity.

2.2 Software System and Component Description

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 the software system description is to provide enough semantic information about a particular software system so that it can be deployed in an automated fashion. The responsibility of describing a software system lies solely with the producer of the software system since it has the most knowledge about the system.

We have identified five classes of semantic information that must be described for a software system or component, these are outlined in the following subsections.

Assert constraints: The correct operation of a software system is dependent upon values of properties at the site where the software is to be deployed; these types of constraints are assertions and they cannot be resolved in the presence of a conflict. Some possible examples of assertions are the requirement that the operating system be Windows 95® and the screen resolution be greater than 800 pixels by 600 pixels.

Assertions 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 assertions 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 second use of assertions is best described with an example. Imagine a particular deployed software system that requires version 1.0.2 of the Java Virtual Machine. This constraint was, by definition, true at the time of installation, but anytime after installation this constraint may be violated if the Java Virtual Machine is upgraded. The result is that the deployed software system will no longer function properly since its constraint on the Java Virtual Machine has been violated.

Assertions can be used to 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, assertions must be maintained and managed at the consumer site, thus a consumer site description is essential for deploying software systems.

Complications may arise from these inherited assertions. For example, a deployed software system may be constrained by version 1.0.2 of the Java Virtual Machine. A new software system that is being deployed may be constrained by version 1.1.4 of the Java Virtual Machine. 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. Using assertions, such a situation is guaranteed to fail even though it might be possible to install multiple versions of the Java Virtual Machine. This last point illustrates the need for treating dependency constraints distinctly from assertion constraints.

Dependency constraints: Dependency constraints have a means for resolution if a conflict 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 can be found in a Web page-based software system. This type of system depends on the existence of a Web browser in order for it to operate. It is very likely that a Web browser could be retrieved and installed if one does not exist, thus subsystem dependencies are one type of dependency constraint.

The line dividing assert constraints from dependency constraints is subjective, though. Recall the previous example of two systems requiring different versions of the Java Virtual Machine. In this scenario the second software system being deployed could retrieve and deploy its required version of the Java Virtual Machine since the two versions of the Java Virtual Machine do not conflict with each other. Since the Java Virtual Machine consumes a relatively small amount of resources it may make sense to allow multiple copies. In other scenarios, though, multiple copies of a subsystem might not make sense because of incompatibilities or impracticalities.

These scenarios emphasize the importance of having distinct notions of an assert constraint and a dependency constraint; assert constraints only assert that the constraint must be 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 be resolved by simply requesting that a new copy of the subsystem be deployed to meet the new constraint specification. It is clear that this is not a desirable situation; the resulting consumer site would be littered with various versions 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 may be possible to attempt to reconfigure the functional aspects of the dependent subsystem. As another example, a dependency constraint may be placed on the configuration of the consumer site operating system. In such a case it might be possible to resolve a conflict by reconfiguring the operating system. The notion of a dependency constraint must be flexible.

Dependency constraints are also complicated because there are different types of dependency constraints. By expanding beyond the notion of corequisite dependencies, a class of temporal dependency constraints is found. Temporal dependency constraints exist at 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 being 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 may be removed after the software system is installed without affecting the proper operation of the software system.

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 should be considered separately.

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. This information may include the location, description, and type of the artifacts included in the software system.

Configuration: There are two types of configuration information that describe a software system or component. The simpler of the two is the settings or configurable properties of a software system. A software system can generally be configured in many ways. The various configurations may determine simple aesthetic issues or they may be used to determine various levels of performance or functionality. It is therefore imperative that the variability of the software system configuration be described so that it can be examined and possibly changed by other software systems

that, in some way, depend on the particular software system. 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 regards 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 can be partly thought of as 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 the server must be activated before the client can be activated. In addition to relationships between components, the interfaces and capabilities provided by those components must also be described. 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 can be 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.

Activities: Despite the fact that the previous classes of semantic information describe a large portion of a software system for deployment, it is still not possible to know all the specialized activities that need to be 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. This type of activity cannot be easily inferred from the previous classes of semantic information and must be described separately.

No matter how well thought out a deployment language or schema is, it will not be 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.

3 Conclusion

The use of networks, such as the Internet, to distribute software to consumers is proving to be very beneficial for both the consumer as well as the software producer. In order to realize the potential of network software distribution, though, the definition and standardization of software system and component description must be

introduced. Describing software systems and components in a complete and rigorous manner is required for the creation of a general infrastructure to support the software deployment life cycle. Such a software description definition must include ways to describe system assert constraints, dependency constraints, artifacts, configurations, and specialized deployment activities. Combining such a definition with a semantic description of consumer sites makes it possible to create general solutions to the various deployment tasks.

Further effort in these areas is being researched by the University of Colorado's Software Dock [3] project. The main purpose of this project is to create a standard, rigorous schema for describing software systems and consumers sites and to create a software deployment framework to utilize these descriptions. The approach taken by the Software Dock is to develop a distributed framework where various agents are used to interpret semantic deployment information and to then automate the software deployment life cycle tasks.

Acknowledgements: This work was supported in part by the Air Force Material Command, Rome Laboratories, and the Defense Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

1. Desktop Management Task Force, Software Standard Groups Definition, Version 2.0, November 29, 1995. (http://www.dmtf.org/tech/apps.html)
2. J. Estublier, R. Casallas. "The Adele Configuration Manager," Configuration Management, Wiley, 1994, pp. 99-134.
3. R. S. Hall, D. Heimbigner, A. van der Hoek, A. L. Wolf. "An architecture for Post-Development Configuration Management in a Wide-Area Network," Proceedings of the 1997 International Conference on Distributed Configurable Systems, IEEE Computing Society, May 1997, pp. 269-278.
4. R. S. Hall, D. Heimbigner, A. L. Wolf. "Software Deployment Languages and Schema." Technical Report CU-SERL-203-97, University of Colorado, Dec. 18, 1997. (http://www.cs.colorado.edu/serl/cm/Papers.html#Schema)
5. A. van Hoff, H. Partovi, T. Thai. "The Open Software Description Format (OSD)," Microsoft Corp. and Marimba, Inc. (http://www.w3.org/TR/NOTE-OSD.html)
6. B. R. Schmerl, C. D. Marlin. "Versioning and Consistency for Dynamically Composed Configurations," Proceedings of the 1997 International Symposium on System Configuration Management, Springer, 1997, pp. 49-65.
7. E. Tryggeseth, B. Gulla, R. Conradi. "Modeling Systems with Variability using the PROTEUS Configuration Language," Proceedings of the 1995 International Symposium on System Configuration Management, Springer, 1995, pp. 216-240.