From the Proceedings of the 6th Inter. Conf. on Software Configuration Management, LNCS 1167, Springer-Verlag, Berlin, 1996

Post-Deployment Configuration Management

Dennis Heimbigner and Alexander L. Wolf

University of Colorado, Boulder, CO 80309-0430, USA

Abstract. Configuration management must move beyond its traditional development-time concerns with source files. Future software systems will be delivered in binary form and will be composed from pre-existing systems. The term Post-Deployment Configuration Management is proposed to address the problems introduced in managing these new kinds of versioned configurations.

1 Introduction

Traditionally, configuration management (CM) research has supported the management of development-time configurations. As a rule, these are configurations of the source and binary files comprising a software system as it is developed. A tacit assumption here is that the development configuration covers the whole of the deliverable system.

Increasingly, however, delivered software systems only cover a small portion of the software that must be composed in order to construct the total system and make it ready for use. For example, a given piece of software is often only partially instantiated at delivery and must be completed by combining it with such things as shared libraries. Further, even a relatively self-contained piece of software must be parameterized to meet local conditions. In the Unix world, for example, it is common to have to establish pointers to appropriate local file paths by either recompiling pieces of the software or defining environment variables with those file paths.

We use the term **post-deployment configuration management** (PDCM) to refer to the process of taking a delivered, but incomplete system and installing, configuring, parameterizing and instantiating it for actual deployment in the field. We believe that PDCM is sufficiently important to bring it out of the shadow of the broader term "maintenance", where it is often hidden.

As distributed object technology takes hold, there will be an additional problem in establishing connections between various distributed objects Although, PDCM is an issue for almost any kind of deployed software system, we have been focusing on its role in distributed object systems, so we generalize it somewhat to include specifically the managing and installation of remote (networked) component-based, versioned software across heterogeneous environments.

2 Post-Deployment Tasks

The post-deployment configuration process is surprisingly complex. In outline, it covers the following high level tasks:

- *Installation* refers to the problem of installing the source and/or binaries of a software system into the local file system. This includes the problem of managing the dependencies between installed systems.
- *Parameterization* refers to the modification of a software system to take into account the local site context. This includes such things as the location of libraries, various file system paths, and hardware.
- *Instantiation* refers to the process of taking the system binaries instantiating them as, for example, Unix processes, interconnecting them appropriately, and starting the system into execution.
- *Re-configuration* refers to the problem of statically and dynamically reconfiguring the system to evolve it over time. This process may involve re-installation, re-parameterization, and re-instantiation.

We are addressing specific problems within each of these areas.

3 Installation

Installing a software system can be a difficult task. Not even the physical placement of the files into the proper locations in the file system is always simple because various sites will have quite different local standards about where software should be placed. To further complicate things, if the site is heterogeneous with respect to operating systems and hardware, then there is often a conflict between the assumptions built into the software and the assumptions at the local site. Examples include handling of shared files (i.e., those common across all hardware platforms) and the proper locations for libraries versus executable binaries.

Most major software systems are dependent on many other related software resources. This seems especially true for software available over the internet. These dependencies can take several forms. Consider, for example, building the system from the source. This can require a variety of tools, and often specific versions of tools (gcc version 2.6.2 versus gcc version 2.7.1, for example). Or, consider run-time dependencies where a software system depends on the availability of other software resources in order to execute. Examples might include dynamic libraries, various operating system files, and compute servers.

Thus to build any one system, it may be necessary to build a number of supporting systems, which in turn need other systems. In other words, the support structure *cascades*. Managing, or even determining the resulting dependent software can become a major overhead on the original installation task . We also note that the problem is complicated if (1) the local site already has some of the dependent software installed, and (2) if ranges of versions of dependent software can be used (i.e., version 3.1 through version 3.3 of Q [3]).

We are addressing this problem by constructing a release manager on top of NUCM [6]. This release manager can maintain dependency information and help bring in proper versions of all required software.

4 Parameterization

Software is usually parameterized to take local context into account. Parameterization can encompass almost any site-dependent modification. We try to separate parameterization from the version dependency problem and from the specification of structural elements, which we intend to be part of instantiation. But of course we recognize the fuzziness of these separations.

The current state of the art in parameterization (at least for Unix) is such things as environment variables, *X* resource files, and *Autoconf* [2].

Environment variables essentially define a simple database of (name,value) pairs where both the name and value are strings. Programs can query this database to pick out local values for such things as file paths, hardware (i.e, display names) and such. Attempting to encode any form of complex information into an environment variable is a frustrating experience.

X resource files improve on environment variables in a couple of ways. *X* resources provide a tree structure for parametric information and provide a primitive abstraction mechanism (called, unfortunately, "classes"). The *X* server serves as the database for storing resource information. Basically, the *X* server stores a forest of trees where the leaves are values and can be tagged with a limited set of types. The interior nodes of the trees have no necessary semantics, but by convention at least the top node represents the name or "class" of a particular program. Again, programs can query the database, but the queries are somewhat complex in allowing, in effect, use of the regular expression "*" operator to specify values independent of their depth in a given tree.

The *X* mechanism is a clear improvement over environment variables, but we believe that it has some deficiencies.

- The abstraction mechanism seems to be woefully underpowered, and in practice, it seems like each class ends up having only one member.
- It would be desirable to compute resource values on the fly. For example, there might be a unique identifier resource that changes every time it is read.

Autoconf is a program used by most Gnu software systems, and occasionally by other systems, to parameterize software sources before compilation. One typically brings in a tar file and expands it. Then one goes into the subtree and executes a *config* script that interrogates the local system for information and produces a *Makefile* [1] for building the software system and taking the local context into account. The *config* scripts do all sorts of amazing things to find out local information: they may run the local C compiler with various test programs, for example. *Autoconf* is actually the program that is used by the software developer to construct the *config* script. *Autoconf* uses a specification of parameters taken from the fixed set known to *Autoconf*. For each specification element, it inserts a corresponding test into the *config* file to be executed at installation time.

We believe that the important idea hiding in *Autoconf* and *X* resources is that of database about the local system. We view, for instance, the Microsoft Win95

Registry system as a rudimentary form of the desired capability. In effect, every machine should be running a server that can be interrogated to extract the kinds of information now known to X servers and to config scripts. Additionally, this server should be dynamically programmable to handle such cases as the unique id example.

5 Instantiation

Instantiation is the process of taking the system binaries instantiating them as, for example, Unix processes, interconnecting them appropriately, and starting the system into execution. We also include the shut-down problem as part of this as well.

Instantiation is where our assumption that we are dealing with collections of distributed objects really has consequences. In the case of a simple, single binary executable, the instantiation process is not particularly difficult. But when one needs to instantiate a number of servers, in the correct sequence, and correctly interconnect them, and connect them to previously instantiated servers, the problem becomes much more difficult.

This problem has been addressed in the case of what we would call "distributed computations." Polylith [5] is perhaps the canonical example here. By the term "distributed computation" we mean that there is a program that happens to be decomposed into distributed subpieces and whose purpose is to start up, complete some computation, and shut down. We distinguish the distributed computation case from our case in which we are setting up services that are accessed by other anonymous programs that might not even be running at the time the services are instantiated. Again, we recognize the overlap and that many of the solutions used in the distributed computation case can apply in our target situation.

Instantiation is most clearly a configuration problem in which one specifies the components to be instantiated and the ways they are interconnected. But there are complexities to be dealt with. The number of servers may need to be computed at start-up time based on various local resources such as performance or available instances of certain kinds of hardware. Additionally, the interconnections may need to be dynamically computed, and moreover, some connections may need to reach to presumed pre-existing services that need to be named in some way. An example might be the assumption of the existence of a database server in the environment to be used by the services being instantiated.

One surprising way that this kind of information has been handled is using X resource files again. Instead of using the resource trees to specify parameters, they have been used to specify structural information. Thus, a program might start up, read the resource file and use information there to instantiate and connect subcomponents. While X resources are not particularly good for this kind of specification, the capability does indicate a close relationship between parameterization and instantiation.

Most existing configuration specifications are static in the sense that they are embodied in text files that are processed at instantiation time. As with parameters, we believe that this information need to be kept as a service where it can be queried and modified dynamically. The OMG CORBA [4] model includes a simple instance of such a server. But again, we believe that this server must allow for programmable specifications which, so far as we are aware, is a capability not currently provided anywhere.

6 Re-Configuration

One a system is instantiated and executed, it will evolve over time. This evolution will be both dynamic and "static" (not perhaps the best term). In the dynamic case, one may need to may limited changes in the instantiated configuration while the software system is executing. These kinds of changes might include upgrading some particular server or changing the number of instances of some server or modifying the interconnections. The static case involves taking down the system and looping back to some previous step to change the installation, parameterization, or instantiation of the system. We are just beginning to address the re-configuration area and hope to build on the technology we develop for installation, parameterization, and instantiation, as well as enhancing our own distributed object technology (Q [3]) to support dynamic reconfiguration.

Acknowledgements. This material is based upon work sponsored by the Air Force Materiel Command, Rome Laboratory, and the 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. Stuart I. Feldman. Make – a program for maintaining computer programs. *Software – Practice and Experience*, 9:255 – 265, 1979.
2. D. Mackenzie, R. McGrath, and N. Friedman. *Autoconf: Generating Automatic Configuration Scripts.* Free Software Foundation, Inc, April 1994.
3. Mark Maybee, Dennis Heimbigner, and Leon J. Osterweil. Multilanguage Interoperability in Distributed Systems. In *Proceedings of the 18th International Conference on Software Engineering.* IEEE Computer Society, March 1996.
4. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 26 August 1991.
5. J.M. Purtilo. The polylith software bus. Technical Report CS–TR–2469, Department of Computer Science, University of Maryland, College Park, MD 20742, 1990.
6. André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf. A Generic, Peer-to-Peer Repository for Distributed Configuration Management. In *Proceedings of the 18th International Conference on Software Engineering.* IEEE Computer Society, March 1996.