Beyond Source Code Control: Object Driven Software Configuration Management

Introduction

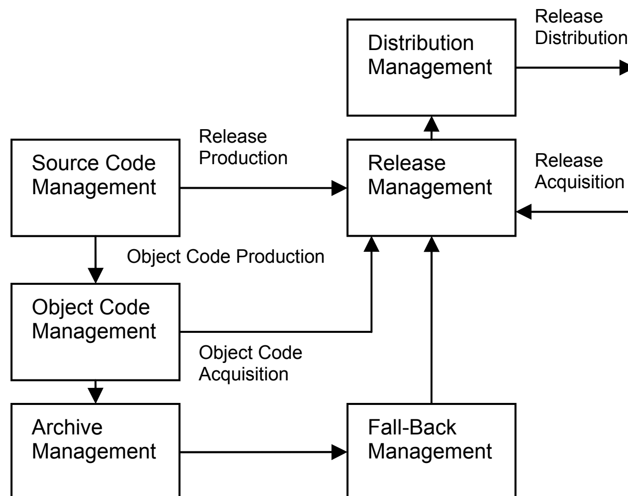
In the world of NonStop and mission critical systems, the terms "revision management" and "change control" have been in the lexicon of system managers and developers for many years. But what do these terms mean? Depending on the group you speak to, very different views of the scope of these two terms come through. Generally, developers view revision management as pertaining to source code, while production managers think in terms of maintaining the integrity of executable objects.

The methodology used in the creation of PrimeCode/RMS seeks to demystify the general use of these terms. A revision is simply a change. Hence, the management of revisions refers to the management of change. In a system that tracks revisions, one can produce a history of changes. The term "revision management" as applied to development and production computing environments describes the process of controlling and tracking all changes in a system.

The strategy of an object-based revision management system is geared towards the faithful reproduction of program objects. It is a requirement of the reliable functioning of a production environment to have accurate and complete reproducibility of object code.

This paper describes the problems of revision management in general and specific to the NonStop computing environments. This paper is not a comparison between any two existing products rather it is a presentation of the issues involved in revision management and the importance of a structure like that of PrimeCode/RMS in dealing with those issues.

Figure 1: Ancestry of Object Driven Revision Management



Development Environment

There are four (4) main types of development environments, all varying in complexity and all varying in their Software Configuration Management needs.

I) The Simple Development Environment

A relatively straightforward organizational structure is the in-house development shop. In this set-up, the same systems personnel are responsible for producing and maintaining the source code, producing and running the executable objects, handling technical support issues and fixing product errors (bugs).

A variation of this structure has a "production" group that manages the "live" programs. This structure, however, more closely resembles a client-vendor relationship.

II) The Client-Vendor Relationship

In client-vendor relationships, source code is not generally an issue. The vendor delivers, in machine-readable form, a set of executable programs that perform certain functions for the client. When there is a problem, the client calls the vendor for technical support.

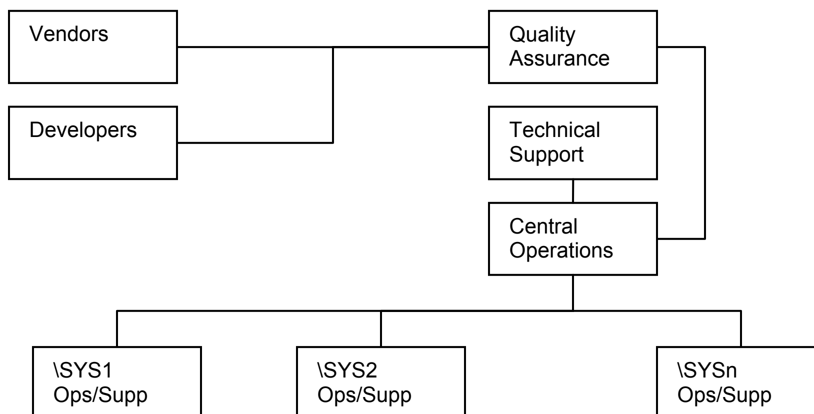
In a minor variation of this arrangement, the client has possession of or access to a copy of the source code that is only to be used under pre-specified conditions (e.g., the vendor has filed for bankruptcy).

III) The Distributed Computing Environment

Distributed environments generally operate in a manner similar to client-vendor arrangements. There is typically a development machine (node) and number of production machines.

This situation invariably arises in which nodes running the same programs have differing versions.

Figure 2: The Distributed Computing Environment



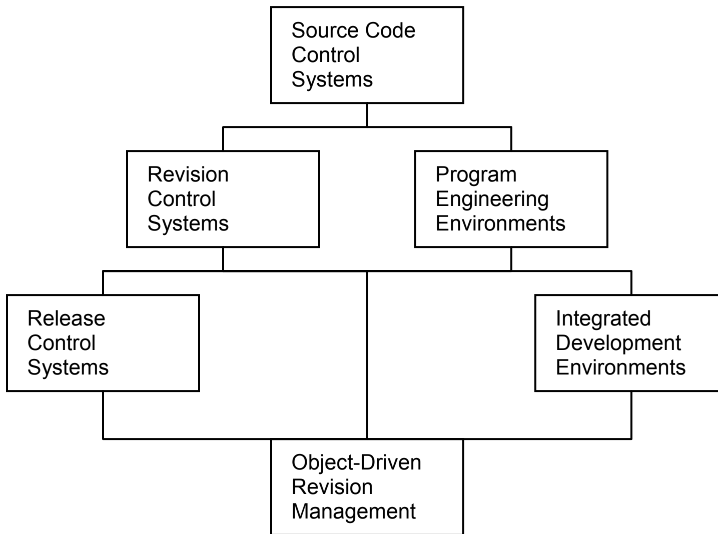
IV) Non-Uniform Environments

Distributed systems, even when operating the same versions of software, often have different system configurations. For this reason, a program may function correctly on one nod, but not on another. There may also be multiple copies of an application environment on a single node. A configuration conflict may exist between two such environments.

Object-Based Revision Management

Object-based revision management refers to the tracking of changes in a system relative to the production of program objects. In this view, revision management is driven from the requirements of the production environment rather than from those of the development environment.

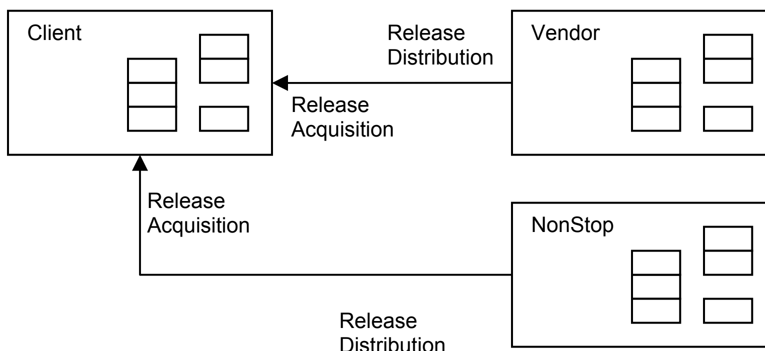
Figure 3: Revision Management Structural Overview



Object-based revision management includes the following issues:

- Management of the history of changes of source code components. This includes enhancements as well as bug-fix modifications to source code. Source code components can also include document files, DDL source and libraries, and EDIT-format configuration and procedure files.
- Production of object files from source components.
- Management and control of releases to clients/other groups, including all components necessary to construct a complete operating environment.
- Management of external, vendor-supplied object code components and data files. These components are not reproducible from source programs within the client's environment.
- Distribution of release to clients/other groups
- Archival and maintenance environment fall-back libraries
- Security facilities

Figure 4: Revision Management Systems Interrelationships



Source Code Management

Source Code Management systems have been around for many years. One of the first such systems to gain wide acceptance was the UNIX-based Source Code Control System (SCCS). Such systems are generally good at managing the differences between versions of sources. One is permitted, under these systems, to extract any version of a source component, and optionally modify and resubmit the changes to the system as a new version. Generally, there are no provisions in source code management for object file production or reproducibility.

Object Management

Object management involves the faithful reproduction of program objects from program sources. Traditional source code management systems such as the Revision Control System (RCS) from Berkeley required augmentation in the form of program production software. The solution used under UNIX was the standard MAKE facility. MAKE is a program production utility that, given compilation instructions and dependencies, determines and executes the least number of operations to create a set of program objects based on source file modification time. Utilities of this sort, however, do not have the ability to recreate past versions.

Release Control

Release control systems have, for the most part, been manual (i.e., paper-based). Personnel are assigned to create a bill of materials or “release manifest” (discussed later) based on input from the developers. Documents and manuals are produced that describe the contents and installation instructions, which may be out of date by the time the release actually occurs.

A tape or subvolume is then produced with whatever is current in the development system. This can sometimes be awkward if changes are being made while a release is being produced. Adding something to a release can be a bureaucratic nightmare.

Non-Reproducible Component Management

An unfortunate reality of revision management is that there are inevitably components that are either not reproducible (i.e., no source files) or are simply data files. To ignore these files allows uncertainty to enter into a release. Revision control systems should be able to handle changes to such files either through difference processors (similar to those used in source code management systems) or by simple archival methods.

Distribution Control

Software distribution control is actually a two-phase management problem. The first phase involves the production of a release suitable for shipping (e.g., on a tape or over a network). The second phase involves the receipt of the release.

The difficult part of the first phase is actually generating the release (normally handled by a release control function). A release manifest is then produced in a format suitable for the site where the release is being sent.

The second phase involves the integration of the incoming release with the target site’s local revision management system. If source is not provided, then the object components can be treated as non-reproducible components. If source is provided, object components should be produced directly from the source rather than from the release components to ensure the integrity of the release. In either case the release manifest directs the introduction of the release into the local revision management system. Currently, this phase is handled using a variety of administrative solutions ranging from maintaining full tape backups of all release components to the simple overwriting of the existing software. NonStop™’s INSTALL facility improves the reliability of release integration.

Archival Facilities

Archival facilities are key to revision management systems. These facilities:

- Allow the retrieval of a fall-back library if a production release has a bug that cannot be circumvented;
- Allow developers to roll out changes that are inappropriate;
- Facilitate the maintenance of historic records

Security Facilities

Security facilities are important to virtually all production systems. It is critical that only that which is permitted to be in production actually gets into production. For example, we wish to prevent insertion of Trojan horses in to production programs. Some software (e.g., cryptographic routines) may actually be hidden from all but key personnel. A revision management system must take these and any other items discussed later into account if software is actually to be used in production.

The Development Environment

One of the fundamental tenets of computer system design is that programs must be usable. This is especially important in the development environment. Programmers often are less than tolerant regarding controls placed on their environment. A development environment which uses a form of revision management should contain all of the appropriate controls (discussed later) but permit the developers to function effectively.

The Development View

The development view refers to all facets of system development excluding the actual managing of the programs in production. This includes testing, quality assurance, and technical support phases ot the software life cycle.

Developers need to be able to function effectively in the creation, maintenance and compilation of software components. This implies either:

- that the revision management system must be able to fit into the native environment with little impact;
- that the native environment must be able to handle the revision management system with transparency; or
- that a completely independent environment for development must be established.

In the NonStop context, a revision management system must be able to handle and produce EDIT-format files, plus interact correctly and completely with compilers (such as TAL, pTAL, JAVA, COBOL, SCOBOL, DDL) and the link editor (BINDER). This implies that the source components must be presented in a manner acceptable to the native language processors. The system must be able to identify versions of the object files produced by these compilers.

Prior to delivery to either a testing or quality assurance group, the revision management system will need to obtain rules regarding the production of object files (what source components are used and how they are used). These rules should be established by the developer. Following delivery to a client, there should be no changes to the delivered source code in order to implement the client's installation.

Aside from any source code management issues, the developers must also keep in mind that environment in which their software will run when in production.

The following case study illustrates a problem which can arise when software is managed without being mindful of the environment in which it was designed to run.

> Software Technology Inc. put together a high-throughput communication system using NonStop™'s 6100 communication subsystem. In two years of production (100% uptime) at a client's site there had never been any report of problems with the software. The client, requiring more communication lines, linked up two additional systems which had existing preconfigured lines. The system still functioned correctly on the original lines. For unknown reasons, there were occasional transient problems on one of the new systems, while the software did not work at all on the other. What no on had told that client was that the software would only run on a 6100 using specialized CLIP code. The first machine had that standard code, with the other used an older controller.

The Production View

The production and audit groups require information relating to the release manifest (discussed in next section). The information in the release manifest must be sufficient to answer the following questions:

- What has changed?
- Who made the changes?
- What effect has there been on object code, if any?
- Has there been an actual change to the source or have all changes been comment-related?
- What compilers were used?
- What operating system and version were used for the compile?
- How did we get to this version?
- Are we following a legitimate upgrade path?
- What versions are actually of should be in production now?
- Can we fall back?
- What are the steps needed to fall back?
- Have there been any code patches, and if so, what are they?
- Can we reproduce all components exactly?
- Are the objects that we have in production actually the objects that we are expected to have?
- Are the source components faithfully represented by the object modules?

These questions effectively all boil down to one: "What do we have?" Another case illustrates the important of a release manifest:

> Widget Corporation has been running a production control system for four years. There have been two major releases and some number (unknown) of bug fixes and code patches for the software following the initial delivery. A computer virus was inadvertently introduced into the production environment about eight months prior to detection (at least that was the system manger's guess since backups are kept for only eight months). The programs were also compiled using aversion of the copier what had knows bugs which were exploited by the programmers. After three weeks of trying to recreate the production objects, the system manager was fired.

The Release Manifest

The relationship between a client and a vendor or between a production group and a development group are similar. The delivery of object code and handling of technical support issues are generally the responsibility of the vendor of development group. The acceptance of the code, ongoing management, and problem reporting are generally handled by the client or production group. Adequate, mutually understandable documentation regarding the correct contents of each release is key to problem resolution. Even if the client does not need to see the actual source versions that created a specified object, the vendor must have the documentation available.

This documentation is known as the release manifest. A specialized form of the release manifest which deals only with delivered components is known as the bill of materials.

A release manifest completely describes the composition of a release. The following is a summary of the contents of a general release manifest (note that some environments may require items not mentioned here):

- General release information, release name and version, description of the release
- Brief summary of functional differences
- Reason for the release
- List of TPRs or user change identifiers associated with or corrected by the release
- List of object components and respective version identifiers
- A difference listing of all changes to objects associated with the release (could be derived)

If source is provided:

- List of source components and respective version identifiers
- Dependencies between object and source components
- Dependencies between object components
- Dependencies between object components and compiler/operating system product versions
- Complete compilation instructions
- A difference listing of all changes to sources associated with the release

If code patches are provided:

- List code patch components
- Dependencies between objects and code patches
- Dependencies between code patches to deterring the correct order of patch installation
- Range of valid operating systems that are permissible for execution
- List of valid “prior” release which must be in place in order for the installation of the release to proceed
- Installation instructions if the release needs special installation procedures
- De-installation instructions if the release needs special procedures to effect a successful fall-back

Client Use of the Release Manifest

A computerized business usually depends heavily on correctly functioning software. Software generally works only when it is installed correctly. A fall-back usually involves a reinstallation of the software from a backup with data that is compatible with the software. A manifest delivered with the software should describe all pieces necessary for the correct function of the system even during a fall-back situation. In addition, the manifest should really indicate a trap-door situation where fall-back is not possible.

In the situation where a client is receiving software from a vendor, the client would, upon receipt, place the delivered software in a properly managed environment. This environment would keep track of the delivered objects in archival form. It is important that the fallback release, if any, be available for re-creating in the event of a critical failure in the current production release.

Vendor Use of the Release Manifest

Vendors who have technical support arrangement with their clients need to know exactly which version of a particular piece of software is in use at a given client site. Without this knowledge, problem resolution can be as difficult as looking for a needle in the wrong haystack. The release manifest should apply for all releases of a product and any bug fixes or enhancements. Ideally, a vendor should

be able to provide a release manifest which is compatible with the client's revision management system. Knowing which releases contain or are derived from a specific changed source component can be used to determine where upgrades need to be sent.

The Link between the Client and the Vendor

If a client and a vendor share the same revision management environment (e.g. they work for the same organization. In this scenario, the vendor is the development group, the client is the production group.) release manifests are easily generated for both parties. The production group can see exactly which components went into a particular program. The development group can see exactly which programs, at any level, used a particular source component.

If the vendor is receiving source with the object code, as a result of either a source license or a contingency clause in a contract, it is critical that the source code and object code be rationalized. The first requirement of a build is a guarantee that the build is complete. A tool must provide a means of confirming that the release is complete in all of its parts before delivery. In addition, the release manifest must describe the linkage of source and object code in enough detail for the client's revision management environment to be able to faithfully reproduce the object code.

The following case study is offered as an example of a release build from a system that did not detail these critical linkages. As a result, the release was incomplete and lacked the necessary instructions for reassembly.

> A large multi-national bank purchased a software package containing some 400 programs with an arrangement to keep the source code in escrow in case the vendor went out of business. After six months of running in production the bank was notified that the vendor filed for bankruptcy. This did not immediately alarm the client since it had received all upgrade tapes for the sources. However, none of the sources for any bug fixes and customized enhancements had been received; nor had any compilation instructions. The development team found that the expected functions of the objects could not be reproduced with any reliability. Two months after the vendor filed for bankruptcy, the bank decided to scrap the programs and associated hardware.

Both of the critical requirements failed to be met as the escrow tapes were delivered in an incomplete condition and lacked the necessary detail concerning the relationships between source and object files to be useful.

<u>Distribution Mechanisms</u>

Distribution management involves the creation, delivery and tracking of a system which is being sent somewhere. "Somewhere" may be a production subvolume on the same node, another node or a completely separate system or network. Traditional distribution mechanisms include magnetic tape, EXPAND links, and file transfer protocols or a specialized distribution solution. Regardless of the actual mechanism, it is critical that a revision management system ensues that all necessary components and only the desired components are distributed. It is important to know exactly what went out the door.

It is equally important to know exactly what has been received. The vendor has no way of knowing whether a client has taken a distribution tape and placed it in the library without installation. To assist in adequate problem resolution, a client who has responsibility for his own system must keep track of the history of change for each application. Operators should keep operator logs to track configuration changes and system problems. System managers should keep track of what was installed and when.

<u>Configuration Management</u>

Configuration management is an important issue in product system control. There are invariably differences in the manner in which systems are configured. The ability of an organization to handle such differences is critical.

Version Differences between Environments

A frequent scenario encountered in multi-node distributed environments is as follows:

> A new release of a product has been prepared and passed through all quality assurance testing. The decision has been made to take the release to production one node at a time over a number of weeks. Two weeks into the procedure, there is one production node at the "older" level, two production nodes at the "release" level and as there is ongoing development, the development node is at a "newer" level.

A technical support person handling a production problem must be aware of the specific version of all relevant sources which made up the objects which are running on the node that had the problem.

Configuration threads are designed to handle such situations. These threads allow the release managers to bind specific versions of sources to a release or environment. A technical support person can thus obtain a complete picture of the program versions using that information.

Another problem relates to the "Minimum/Maximum Acceptable Version" of a component. It is perfectly reasonable to expect to be able to request version 1.003 of a component SRC and version 1.006 of component DDLTAL assuming that those versions are compatible. What is the minimum acceptable version compatible with SRC:1.003 is DDLTAL:1.010? DDLTAL:1.010 could represent a trap-door. Once a production node used a release containing 1.010 of DDLTAL, fall-back to a previous release would require rebuilding the data in the database from the point in time when the release was installed.

Content Differences between Environments

The handling of variations in environment-specific software is important in multi-environment systems, as the following case study points out:

> A large financial institution was running three copies of the same PATHWAY application in separate environments. A requirement came forward to have another six copies of that application running in a two-month time frame. Unfortunately, the development group found over 4000 independent parameters that had to be changed for each environment. Some of these parameters were contained in non-EDIT-format files.

The above is an example of content differences between environments. These differences are generally handled using either branching or parametric techniques.

Branching involves the tracking of changes to source components in parallel. This is most commonly done by vendors who are tracking significant code customizations between clients. A particular change could be applied to all or some of the branches depending on the scope of the change.

Parametric configuration involves the use of any combination of run-time parameters, macro pre-processors and automatic file construction/modification programs.

<u>Security Considerations</u>

Although some environments permit developers and support personnel free access to virtually all elements of a production environment, most production shops require some security. There are many possible levels of security and access controls that can be placed on components.

Source Code Security

Source code security is an essential part of access control. Source code security describes the manner in which developers can use a component of source code.

There are two aspects to access control: permanent and non-permanent access. Permanent access control describes the ability of a user to see and/or modify a particular component at any time. This type of security is generally handled by access control lists or basic security attributes.

Non-permanent access control involves the use of locks. Locks temporarily remove modification access to specific versions of components while they are being modified. This type of access control is standard to most source code management systems. Some systems provide a modified form of locking which permits multiple designated users to have modification access to a given component.

Object Code Security

Object code security is generally handled by the standard operating system security. The only time that this is really needed is during release production. At this time, it is critical that only authorized users produce release objects.

Logical Version Security

Software management environments may be roughly divided into the following distinct groups:

- Development
- Testing and quality assurance
- Production support
- Operations

Revision management systems frequently allow software to be labeled with states corresponding to such an organizational structure. Logical version security is then added as a layer on top of any resident security mechanisms. This security layer provides a distinction of access between "states" of source and object components as well as releases. This allows system managers to restrict access for a given group of users to a specific part of the development life cycle.

Release Security

There are a number of items in a release that must be secured during the release production process:

- Contents, in terms of components
- Contents, in terms of versions of components after a release is "frozen"
- Dependencies between components in the release
- Contents, in terms of compiled object components
- Compilation instructions
- Order of compilation
- Installation instructions

Implicit in this list are the actual component contents, which are not permitted to change once a release is frozen. Versions can be added to source components; however, this must not have any effect on the release.

Software Distribution Security

Just as it is important that releases be secured, it is critical that software releases be sent to and only to designated areas. The production sites require that the exact release actually be delivered as specified. Software distribution is the final stage of a full CM process – it creates a secure and managed system for moving releases from development to production or client sites. If a configuration management system does not include software deployment capabilities, it cannot be considered an end-to-end solution.

It would also be highly inappropriate to send a release to a competitor’s data processing site, especially if highly sensitive data and trade secrets were involved.

Figure 5: Revision Management Security Levels

Security Level	Control
Archive Security	Restore Access Control
	Backup Access Control
	Access Control
Distribution Security	Acquisition Control
	Distribution Control
	Access Control
Release Security	Component Version Control
	Content Control
	Access Control
Logical Version Security	State Transition Control
	Access Control
Configuration Security	Component Variant Control
	Parameter Access Control
Object Code Security	Access Control
Source Code Security	Concurrent Access Control
	Access Control

Summary

At present, most of the change management environments that support production systems are managed either by source code management systems, or off-line object archival systems, or a combination of the two.

The future will see all phases of the software life-cycle, from initial product requirements and conception, through development, release distribution and long-term technical support and configuration management handled by an integrated revision control environment.

The release manifest is the central concept of object-based revision management. With a detailed release manifest, communications between clients and vendors during release delivery is accurate and complete. Traditionally, the release manifest has been paper-oriented, requiring substantial effort to produce. As revision technology improves, release manifest production will become progressively more automated.