Version Management in Gypsy

Ellis S. Cohen
Dilip A. Soni
Raimund Gluecker
William M. Hasling
Robert W. Schwanke
Michael E. Wagner

Siemens Research & Technology Laboratories

This paper describes the Version Manager of the Gypsy programming support environment, and its integration with the object-oriented extension of Unix[1] on which it is built.

1. Introduction

The Gypsy Programming Support Environment provides support for a team of developers to produce and maintain systems built from multiple components. The foundation of Gypsy is the Version Manager, which is used to store, organize, and selectively retrieve versions of objects. Integrated with version management are components for workspace, configuration, and event management:

- The Workspace Manager supports task-based software development. It provides a protected environment for working on private task-related versions, and supports authorization and logging on a per-task basis.
- The Configuration Manager supports construction of software products from versions of software components.
- The Event Manager provides support for monitoring changes to objects, and for triggering actions when an event occurs. In particular, it can be used to support tools (such as Marvel [Kaiser87]) that selectively pre-build products as new versions of components are released.

[1]Unix is a trademark of AT&T

More than any other part of the system, the Version Manager is tied to the operating system. Gypsy is built on top of an object-oriented operating system which provides mechanisms that support especially clean integration.

This paper describes the details of the design of the Gypsy Version Manager. We describe its integration with the Operating System and the rest of Gypsy, and discuss some of the integration problems that still remain.

2. Operating System Integration

2.1. Advantages of Integration

The basic file model in Unix does not include any version control facility. That is, if you open a file for writing, close it, and re-open it, you get the same file.

Version control systems like SCCS [Rochkind75] and RCS [Tichy85], which have been implemented "on top of" Unix, are cumbersome to use, because version access and control are not integrated with the file and directory system.

Integrating the version control facility with the operating system, as is done with the Apollo DSEE History Manager [Leblang84], or by using a mechanism such as "Watchdogs" [Bershad88], has a number of advantages:

- Version and file naming are integrated -- the name of a version is simply an ordinary directory pathname extended by a version selector.
- A program or user does not need to explicitly extract or reconstitute a version in order to read it -- a version named as a pathname extended by the version selector can be directly retrieved by the operating system kernel.
- Version management is transparent, so that when an arbitrary pre-existing Unix program opens a file, it does not need to know whether the file is version controlled, or

Recommended by: *Erhard Ploedereder*

whether the file name included a version selector.

- Reconstitution and caching of delta-compressed file versions can be done transparently when needed, when a version of a file is opened. Alternately, reconstitution need not be done at all; reading the opened version can be done by cleverly reading through the delta-compressed representation.
- Versions which are being modified can be automatically hidden from other users without moving the file back and forth between directories.

2.2. Objects and Attributes

Like DSEE, Gypsy is an integrated system. However, Gypsy is built on an extension of Unix that provides mechanisms for arbitrary user extensions to the notions of files and directories. Through the use of these mechanisms, Gypsy version management is provided outside of the operating system kernel.

The operating system is object-oriented, by which we mean that

- Objects are typed. Types may be dynamically created, though some, like files and directories are built-in.
- The code invoked by calling a "method" (a.k.a procedure) is determined dynamically, based upon the type of the object involved.

Objects are accessed by *descriptors* which carry access control rights. When a user retrieves a descriptor from an object, the rights are generally a subset of those with which the descriptor was stored [Cohen75].

However, the kernel also provides direct support for authority-based control [Saltzer75]. If an authority-list is associated with an object, and the object is retrieved from a directory or any *persistent* object, its access rights are limited by those the authority list permits.

The owner of an object can obtain and regain any rights to it. However, only an object's type manager can directly access and manipulate the type's representation [Cohen75].

The binding of methods to code is implemented through *attributes* (similar to *traits* in the software for the Star [Curry84]; see Section 11.1 for its relationship to inheritance). An attribute specifies a characteristic behavior of a type (e.g. that it provides byte-oriented access to its contents, or that it can be backed up and restored). Associated with an attribute is a list of methods used to implement the behavior. Two different types may have the same attribute, but provide different implementations of the methods.

For example, if the creator of a new type of object wants to define byte-stream access to the contents of the object, a BYTE_STREAM[2] attribute would be associated with the new object type. The BYTE_STREAM attribute includes an "open" method.

The "open" method of the BYTE_STREAM attribute for files understands the internal representation for files, and knows how to open a file for reading. The "open" method of the BYTE_STREAM attribute for a different type of object understands the internal representation of that type of object and knows how to open it for byte-stream access.

In either case, a client using an object with a BYTE_STREAM attribute, regardless of its type, essentially just invokes BYTE_STREAM.`open`, and the kernel arranges for the method appropriate to the type to be called. Whichever open method is called returns a descriptor through which the object can subsequently be read (via the type-specific method BYTE_STREAM.`read`).

2.3. Version Groups and Customized Directories

The operating system provides a standard directory type, but the DIRECTORY attribute can be associated with other object types to provide what we call *customized directories*. The DIRECTORY attribute of a customized directory type defines methods for inserting and deleting components, listing the directory, changing protections, and so on.

Gypsy defines a Version Group object type used for managing versions. However the Version Group type is also a customized directory -- that is, it has a DIRECTORY attribute -- and this is used to integrate file and version naming.

For example, suppose that `/usr/glu/shlp.c` is a version group. Then, version 5 of the file on branch B3 of the version group can be named as

```
/usr/glu/shlp.c@B3@5
```

When this name is looked up, the built-in directory manager translates the path name left-to-right, until it comes to a non-standard directory. It then passes the remainder of the path name to the "retrieve" method for that directory, and expects to receive back a descriptor for whatever object was named.

The directory system recognizes `usr` and `glu` as standard directories, and retrieves a descriptor for `shlp.c` from `/usr/glu`. It understands that "@" is a secondary separator in path names (similar to "/"), and that `shlp.c` is a customized, rather than a standard directory. So, the standard directory manager passes the descriptor for `shlp.c` along with the string `"B3@5"` to the DIRECTORY.`retrieve` method for version groups. The "retrieve" method reconstitutes version `B3@5` if necessary, and returns a descriptor for it.

---

[2] Throughout the paper, for the sake of clarity, we will be using names for attributes that may differ somewhat from those actually used in the system.

3. Version Group Organization

3.1. Branches and Versions

Like most recent version management mechanisms, Gypsy organizes versions as a tree. Systems like RCS [Tichy85] use a naming scheme that parallels the branching structure. Perhaps the best decision we made in Gypsy from an ease-of-use perspective was to decouple the naming from the branching structure -- unlike RCS, the name of a version does not describe the complete path from the version to the root of the version group.

Each branch in Gypsy has a name automatically generated by the Version Manager. The first branch created is assigned the name `main`. Subsequent branches are named `B1`, `B2`, `B3`, etc. Versions along each branch are assigned successive numbers beginning with `1`. A version is completely identified by a *version identifier*, consisting of a branch name and version number, for example `B3@5`. Methods are provided for obtaining information about the branching structure -- for example, indicating that branch `B3` branched off from `B1@9`.

Since the built-in names are not particularly mnemonic, users can provide a name for either a branch (e.g. `mybranch is B3`), or for a version within a branch, for example

```
in B3, myversion is 5;
in B6, myversion is 8;
```

and these names may be used in retrieving versions -- for example --

```
/usr/glu/ship.c@mybranch@myversion
```

It would also be useful to have a name correspond directly to a version identifier (a branch plus a version number) -- for example, `favorite is B3@5`, but this is not presently implemented.

3.2. Properties

Properties may be attached to versions; some of them, like author and release date, are automatically supplied by Gypsy -- others may be added by users.

Properties are used to specify information about a version, and can be used to answer queries, or to select a specific version. This is described in section 5.

Properties may be

- an enumerated type -- for example "status", say, with values "untested", "tested", "released", and "integrated".
- a date -- such as the release date set by Gypsy.
- a string -- such as the author property set by Gypsy.
- a version identifier -- a reference to another version in the version group.

We are also planning to extend properties so that they can be attached to branches as well as to versions.

Properties that users or other managers might wish to provide include machine and operating system dependencies, speed, memory or disk usage, and user-friendliness [Narayanaswamy88].

In an object-oriented system, it may seem as though any type of property should be able to be attached to a version, and, through the use of type-specific methods, used in answering queries and in version selection. Though we are considering allowing descriptors for arbitrary object types to be attached as properties, the use of the the *contents* of those objects in queries and version selection would have substantial performance penalties.

The reason is that, in our system, as in other object-oriented *operating* systems [Wulf81,Pollack81], and unlike more general object-oriented systems like Smalltalk [Goldberg83], objects are generally large, and have more than minimal overhead.

To implement version selection efficiently, each individual property of a version is *not* a separate object. Instead, the properties are represented in data structures that are laid out in the memory allocated to the version group object, and optimized for efficient version selection.

3.3. Access Control

As in Adele [Belkhatir86], access to version groups is controlled by an authority list mechanism. In Gypsy, generic support for authority list control is provided as a service of the operating system.

The creator of a version group owns it, and can grant various sorts of access to specified individuals, groups, or to everyone. The creator of the version group determines who can create branches, who can define or change the symbolic names of branches, and who can define properties.

The creator of a branch, however, owns the branch, and can determine who can retrieve versions from the branch, who can lock the branch, who can store new versions on the branch, who can define or change the symbolic names of versions on the branch, and who can set the values of properties of versions on the branch.

Note that individual versions are not separately access controlled. That seemed unnecessary. We did believe though, that versions on different branches should be separately controlled, both for privacy, and to prevent premature disclosure.

The definer of a property also gets to determine who can set that property. In fact, to set the value of a property on a version, permission must be granted by both the owner of the branch, and the definer of the property. For example, the definer of the "verified" property wants to make sure that it is set only by someone who actually is trusted to verify the version. The owner of the branch, meantime, wants to make sure that verification is only done by someone the owner trusts.

4. Workspace Management

The Gypsy Workspace Manager defines a new object type -- a *Workspace* -- to provide an environment for carrying out task-related activities, such as coding, bug fixes, enhancements, and testing. Generally a workspace is created by a single user. It may also be created by a project administrator and assigned to a user.

To work on a task, a user *attaches* to the associated workspace. Attaching to a workspace gives the user rights to private versions protected by the workspace, and sets up environment variables that define the environment to be used within the task.

The Workspace Manager keeps a log of task-related activities including a record of all versions created in the workspace. This log is retained even after the task is completed and the workspace has been *terminated.*

Workspaces isolate other users from changes made inside a workspace. Versions created inside a workspace are *private* to the workspace, and are not publicly visible until they are *released* from the workspace.

Workspaces can also be used to isolate the attached user from changes made outside the workspace. When accessing a version group, a user rarely provides a specific version identifier, but rather asks for some default version, typically the latest one on the main branch (Section 6). When another user releases a new version, the default version may change. The Workspace Manager provides a *version registration* mechanism (Section 6.3) which can make changes to default versions invisible to a user attached to a workspace.

4.1. Workspaces and Private Versions

In non-integrated systems like RCS and SCCS, users typically make modifications to a system by *checking out* new versions of objects, and placing them in a private directory. After making the modifications in the private directory, the user *releases* the private versions to the version group and they become publicly visible.

In Gypsy, users make modifications to a system by *attaching* to a workspace, and creating new *private versions* of objects. These private versions are part of their version group, but in general, are only visible when the user is attached to the workspace. After making the modifications to the private versions (in place, in the version group), the user *releases* the private versions and they become publicly visible.

Because private versions are part of the version group, they can be returned by version group queries and version selection (Section 5), just as public versions can. This can make it easier to specify build rules for test configurations (Section 10).

To create a new private version, a user must first be attached to a workspace, and must then *lock* a branch of a version group. This permits the user to create any number of private versions on that branch, and to release any (or all or none) of them. Locking a branch does not restrict access to released versions on the branch -- it only affects private versions. The branch must be *unlocked* (deleting any remaining private versions which have not been released) before it can be locked by a user attached to a different workspace.

A user can be attached to one workspace at a time, and ordinarily, each workspace has at most one user at a time attached to it. A user can detach from a workspace while it has a branch locked, and later reattach; the branch will remain locked, and the private versions will be retained, though in general, they will only be accessible while the user is actually attached to the workspace (though see Sections 4.4 and 11.3).

4.2. Using Private Versions and Branches

Private versions use a separate numbering scheme. The first private version for a branch is numbered `P1`, the second `P2`, etc., so, a private version may be named, for example, as

```
/usr/glu/ship.c@B3@P2
```

Since private versions on a branch are deleted when the branch is unlocked, private version numbers are reused. Each time the branch is newly locked, the private version numbers on that branch are again allocated starting with `P1`[3]. The use of a separate numbering scheme for private versions prevents holes in the public numbering scheme, which would happen when private versions are deleted instead of being released.

Just like released versions, private versions can be assigned symbolic names and have properties associated with them. When a private version is released, it is assigned the next ordinary version number on the branch; the symbolic names and properties associated with it when it was private remain with it.

Multiple private versions are often used as checkpoints. For example, in a long editing session, a user may wish to save the current state of the object from time to time. Editors integrated with Gypsy may choose to support this by saving the buffer contents to the latest private version, freezing it (to make it immutable and prevent it from being subsequently modified accidentally), creating a new version (which is initialized, by default, to the contents of the last version), and internally relinking the buffer to the new version. At an appropriate point, the user can release a private version (not necessarily the latest one) so that it is accessible to other users, but can keep the branch locked in order to continue development.

It is possible to create private branches -- i.e. branches which branch off from private versions. Private branches are only visible while attached to the workspace and cannot be made visible, even if the version they are attached to is released. However, the versions which are on the private branch can be released to the public branch which was locked.

---

[3]Though this keeps version numbers small, reuse of private version numbers can also cause some confusion. We may change this decision.

Because private versions are part of the version group, and can take part in querying and version selection (when the user is attached to the workspace to which they belong), two additional built-in enumerated properties are defined -- *visibility*, to distinguish public and private versions, and *mutability*, to distinguish frozen and mutable versions.

4.3. Freezing

Released versions are always frozen, and thus immutable. We had considered requiring that all but the latest private version on a branch be frozen as well -- this would have automated a step in the checkpoint model described above, but would have forced the use of private branches in the case where a user was simultaneously modifying two slightly different versions. We saw this as too rigid.

We decided that private versions must be explicitly frozen; they are automatically frozen only when they are released. Even if a private version is frozen, an earlier one can still be modified. Users who are rapidly making changes to two related versions do not need to go through the intellectual overload of creating a separate private branch, though they may do so if they wish. A consequence of this decision though, is that the configuration manager must use timestamps in addition to version identifiers when deciding whether to rebuild derived objects (see Section 10).

Freezing of files is supported directly by the operating system. Other objects are frozen by invoking the type-specific "freeze" method from the FREEZE attribute. If a type does not have this attribute, a copy of the object is made, which replaces the private frozen version. Copying works, because, while various users may have had write access to the unfrozen private version, the Version Manager will not give out write access to the new frozen copy.

4.4. Workspace Access Control

Ordinarily, a private version may be accessed by one user at a time -- the user attached to the workspace in which the version was created. Additional workspace access controls permit concurrent access; however, users who take advantage of this flexibility will have to depend upon synchronization mechanisms outside of those provided by Gypsy (though often, this is informal -- "Hey, are you using ..."). Both close and weak models of cooperation are supported:

- **Close Cooperation:** The creator of a workspace can supply a list of users who are also allowed to attach to the workspace, and can specify whether those users can be attached concurrently. This facility would typically be used when coworkers are collaborating on the same part of a system or in fixing a bug together.
- **Weak Cooperation:** Unlike released versions, private versions can be individually access controlled. The authority list of a private version can contain a list of workspaces (with access rights). Users attached to one of the specified workspaces can access the private version with the specified rights. This facility would typically be used to allow a coworker to get a sneak preview of a needed object before it is released.

5. Queries and Version Selection

A user can obtain complete public information about a version group, including the name of each branch, the version it branched off from, the versions on each branch, and all of their properties. If the user is attached to a workspace, information can be obtained about the private versions accessible from the workspace as well.

This information can be used to

- Answer a Query -- select a *subset* of versions which satisfy some *predicate*.
- Select a Version -- select a *single* version which satisfies some *selector* (or fails if the selector cannot be satisfied).

In this section, we will look first at querying, and then turn to version selection.

5.1. Querying and Integration

We believed that it was desirable to move at least some querying capabilities into the Version Manager -- in fact, some querying capability was required by integration considerations. Consider the Unix shell command

```
ls /usr/glu/ship.c@B3@*
```

Because version naming is already integrated with file naming, one would expect that this would yield a list of the versions on branch B3 of `ship.c`. However, neither the `ls` command nor the shell know anything about versions. Nonetheless, this ought to work by moving filename expansion out of the shell and into directory and customized directory management.

The shell can expand the filename by passing the string `/usr/glu/ship.c@B3@*` to a query method in the standard directory manager in order to get back a list of file names. When the directory manager notices that `/usr/glu/ship.c` is a version group, it can pass `"B3@*"` to the query method implemented by the Version Manager (using the DIRECTORY attribute) which then produces the requisite list of versions[4].

[4]Unfortunately, the version of the shell currently in use just obtains a listing of all the components of the (standard or customized) directory, and then uses its own pattern matcher. Consequently, a separate shell command has been provided which lists a selected subset of versions. This command is needed in any case, since it understands options which control the display of properties and other version-related information.

Since the Version Manager is completely responsible for interpreting the string presented to it, the issue is simply one of deciding what additional kinds of queries are worth supporting, and specifying a language for them. Predicates can be used for specifying both branches, and versions on them.

5.2. Version Predicates

The are a variety of ways one could specify a subset of versions on a branch more selectively than "*".

The simplest case are predicates on version numbers -- e.g. versions before version 4, or the last version, or the last 3 versions.

Another possibility are simple predicates on the properties:

- Versions created, last modified, or released on, before, or after a specific date.
- Versions with a specified author, or versions created in a specified workspace (workspaces are located in directories, and the path name of the workspace is stored as a string property when the version is created).
- Versions with a specific enumerated property value (e.g. all versions with a "status" of "released"), or with a value in a range or in a specified set of values.

For example, the following command should list the tested versions of branch B3[5].

```
ls /usr/glu/ship.c@B3@*[status=tested"]
```

It would be natural to support arbitrary boolean expressions with these simple predicates as terms (as does Adele [Estublier88]), but we did not do so in the initial implementation.

5.3. Branch Predicates

The simplest branch predicate is simply "*[branch]" -- all the branches, so that, for example

```
ls /usr/glu/ship.c@*[branch]@*[status=tested]
```

should list the tested versions on each branch. Once we support properties which are associated with branches, then we will allow simple predicates on those properties to select branches.

In the future, we would like to includelinguistic support for referring to all the branches which have branched off of a given branch (and the branches which have branched off of them, recursively), and for referring to the versions on the path from a branch down to the root of the version group.

[5]For purposes of clarity, the syntax used in this paper differs slightly from that currently used in the system, which, in any case, is still evolving.

5.4. Filter Predicates

Once a predicate has been used to select a subset of versions, it can be useful to further filter that subset based on a property related to the subset. For example, once we have selected all versions on branch B3, we might want to pick those that have the highest status, which we would write as `/usr/glu/ship.c@B3@*[MAX:status]`. Both MIN and MAX values on enumerated properties are supported.

5.5. Version Selection

Because version naming is integrated with the operating system's directory lookup mechanism (Section 2.3), when an arbitrary pre-existing Unix program opens a file, it does not need to know whether the file is version controlled, and can transparently use a filename which may include a version selector -- that is, which selects a single version from a version group. We are familiar with version identifiers (e.g. B3@5) as version selectors. More dynamic version selection -- selecting a version based on properties -- is useful as well.

Selecting a version can be thought of as happening in two stages:

- Using a *predicate* to obtain a subset of the versions in the version group (as in answering a query), and
- Using a *rule* to select one of the versions in the subset

At present, the only rule supported by Gypsy is "latest". If the user can access the private versions on a branch (i.e. is attached to the workspace which locked the branch), then "latest" always considers the private versions (which are ordered by creation time) before released versions (which are ordered by release time). For private versions, "latest modified" would be a useful rule as well. Perhaps "earliest" might be useful also, but we haven't had a good reason to include it.

We denote the conversion of a version *predicate* (which selects a subset) to a *selector* (which selects the latest member of that subset) by replacing the "*" by "latest" (and for branch predicates, replacing "*" by "any"). For example:

```
/usr/glu/ship.c@B3@*[status=tested]
    -- all the versions of branch B3
    with a "status" of "tested"

/usr/glu/ship.c@B3@latest[status=tested]
    -- the latest such version

/usr/glu/ship.c@B3@latest
    -- the latest version on branch B3

/usr/glu/ship.c@any[branch]@latest
    -- the latest version on any branch
```

Note that predicates contain an "*" in order to trigger the shell to do filename expansion. The elimination of "*" in selectors avoids that.

The Gypsy Version Manager does not currently allow a selector to be specified as a conditional sequence of alternates (as in [Mahler88]), which can be tried in turn till a version is found which satisfies one of them.

5.6. Names as Version Selectors

Earlier, we described how a user could associate a name with a particular branch or version. There are times that it would be useful to associate a name with a version selector, which would be re-evaluated on each use. For example

```
in B3, best is latest[status=tested]
```

so that

```
/usr/glu/ship.c@B3@best
```

would always retrieve the latest tested version on branch B3. That sort of binding is not generally supported (except for component binding in configurations -- See Section 10); instead we allow names to be rebound to different versions. So, for example, when a newer version on branch B3 attains the "status" of "tested", the name `best` could be explicitly rebound to it. Indeed, this could be automated through the use of the Event Manager, described in Section 9.

6. Defaults and Registration

Most users are not interested in the detailed organization of a version group. When they want to retrieve a version they probably want the one on the mainline branch (usually `main`), and they probably want the "default" version on that branch -- for example, the latest version, or perhaps the latest tested one.

Similarly, if an ordinary user has been told explicitly to use a specific branch, then they still will not want to track changes to that branch, but just want to use the "default" version on the branch.

6.1. Default Naming

To support ordinary users, a default branch for a version group can be specified, and a default version for each branch can be specified as well. If not specified, the default branch is `main`, and the default version of a branch is the `latest` visible version on it.

When naming a version, the branch name can be omitted to obtain the default branch, and the version number or name can be omitted to obtain the default version, so that

```
/usr/glu/ship.c@B3
```

names the default version on branch B3, and

```
/usr/glu/ship.c@latest[status=tested]
```

names the latest version with "status" of "tested" on the default branch. Defaults can also be used in queries, so that

```
ls /usr/glu/ship.c@*[status=tested]
```

should list all versions with "status" of "tested" on the default branch[6].

To ensure that version selectors are not ambiguous, branch names cannot be used as version names. Consider the following version name:

```
/usr/glu/ship.c@best
```

If `best` is a branch name, this refers to the default version on that branch. Otherwise, this refers to the version named `best` on the default branch.

6.2. Defaults and the LINK Attribute

In our examples, the `ship.c` component of directory `/usr/glu` is a version group object. However, to support users who are completely unaware that an object is version-controlled, we would like the pathname

```
/usr/glu/ship.c
```

to retrieve the default version of the version group (i.e. the default version on the default branch), *not* the entire version group. The standard directory manager supports this through the use of the LINK attribute [Pollack81].

When the standard directory manager retrieves a component from a directory, it checks whether a LINK attribute is defined for that component's type. If so, it invokes the associated link retrieval method LINK.`retrieve`, passing it a descriptor for the component, and expecting to get back a descriptor for a different object to be used in its place. This is used for a variety of purposes within the operating system, including the implementation of symbolic links.

Version groups have a LINK attribute, and the associated "retrieve" method returns the default version. The directory manager does not "follow the link" if the name is followed by an "@", so that `/usr/glu/ship.c@` could be used to name the version group itself.[7]

6.3. Workspace Defaults

A user attached to a workspace, working on a task, often wants to be isolated from changes made by other users. In particular, the user generally does not want the default version of a branch to change when another user has released a new version on that branch, at least without investigating the new version first.

The Workspace Manager provides a *version registration* mechanism to support this degree of isolation. A user attached to a workspace can register (similar to DSEE's version maps [Leblang84]) a specific branch to be used as the default branch of a version group, or a specific version to be used as the default version of a branch.

---

[6] Some special cases: `/usr/glu/ship.c@*[branch]` refers to the default versions on each branch, `/usr/glu/ship.c@*` refers to all versions on the default branch, and `/usr/glu/ship.c@latest` refers to the latest version on the default branch

When a user attached to a workspace selects a version or branch by using default naming, the Version Manager first consults the Workspace Manager to determine whether the default is registered with the attached workspace. If so, it uses the registered default rather than the default associated with the version group itself.

A per-workspace option provides automatic registration. If a default name is used which is not registered with the attached workspace, the version group's default can be automatically registered, providing consistent default naming without explicit user involvement.

Users who register a default version, either explicitly or automatically, may nonetheless wish to know when a new version of the branch is released by another user. This can be arranged with the help of the Event Manager (Section 9). The user, upon being notified, can choose whether or not to re-register the default.

In the future, we may include nested workspaces, especially when Gypsy is integrated with a project management system. Workspaces correspond to tasks, and nested workspaces would correspond to subtasks. Nested workspaces introduce some complications, for example, in handling automatic registration.

We also may include per-workspace registration of names (e.g. `for /usr/glu/ship.c, in B6, myversion is 8`). Names registered in the attached workspace would override names registered in the version group itself. Nested workspaces would then represent a stack of naming environments.

7. Delta Compression and Reconstitution

Gypsy provides delta-compression support for files; the creator of a new type of object can define a DELTA attribute for that type which contains methods for delta-compression and reconstitution[8]. When a version of an object is released, Gypsy sees if that object's type has the DELTA attribute defined. If so, Gypsy calls on the associated method to perform delta-compression. If not, the object is stored as is.

7.1. Delta Control

Gypsy provides fairly fine control over how delta-compression is done. For each branch, it is possible to specify that

- The latest version on the branch be kept in full reconstituted form.
- No versions on the branch be kept in full reconstituted form.
- All versions on the branch be kept in full reconstituted form.

At the very least, we require that the latest version on the main branch be kept reconstituted.

If the latest version on the branch is kept reconstituted, then other versions on the branch are stored as successive backward deltas. If no versions on the branch are kept reconstituted, then all the versions on the branch are stored as successive forward deltas beginning from the version where the branch is anchored. The model is similar to that used in RCS -- see [Tichy85] for more details on this representation.

In addition, specific versions can be kept reconstituted. This does not affect the delta strategy, but can be used to short-circuit reconstitution of a target version -- i.e. deltas are applied from the nearest reconstituted version along the path to the target.

7.2. The Reconstitution Cache

If a version is not already reconstituted, it is reconstituted when the version is retrieved from the version group. To reduce this cost, frequently used versions can be cached in their full reconstituted form.

The maximum number of versions of the version group that are to be kept in the cache can be specified. If a new version has to be reconstituted, and the number goes above the maximum specified, the least recently retrieved version is discarded; other replacement strategies, based on properties, or implemented by calling a type-specific method, could certainly be considered.

Removing a reconstituted version from the cache does not necessarily destroy it. The user who retrieved that version may still be holding onto a descriptor for the reconstituted version, and may even have stored it in some other object (in Unix terms -- made a "hard link" to the reconstituted version). As long as there is an outstanding descriptor for the reconstituted version, it will not be destroyed, nor should it be.

---

[7] Note that the objects stored in a version group can have a LINK attribute as well, but this does not lead to ambiguity with the following naming conventions:

```
/usr/glu/ship.c@B3   -- The default version on branch B3
/usr/glu/ship.c@B3@  -- The link to the default version
                        on branch B3
/usr/glu/ship.c      -- The default version of the group
/usr/glu/ship.c@     -- The link to the version group
/usr/glu/ship.c@@    -- The link to the default version
```

Also note that the versions stored in a version group can be (customized or standard) directories. In fact, configurations in Gypsy are customized directories [Schwanke88b], which allows directory-like naming of built components. Suppose that `/usr/glu/dirgroup` is a version group of directories. Then `/usr/glu/dirgroup@B3@mine` names the directory which is the version named `mine` on B3, while `/usr/glu/dirgroup@B3/mine` names the `mine` component of the directory which is the default version on B3.

There is a problem of ambiguity if the objects stored in a version group are themselves customized directories with a LINK attribute -- for example, version groups. We have not yet worried about special naming conventions for this case.

[8] It also contains methods for combining two deltas into a single delta, and for reversing a delta -- i.e. producing a backward delta given a forward delta (or vice versa).

Unfortunately, once a reconstituted version has been removed from the cache, subsequent retrieval of the version will cause a new copy to be reconstituted, even if the old reconstituted copy still exists. We will discuss this further in Section 11.2.

7.3. Compressing Private Versions

At present, frozen private versions are not delta-compressed. Depending upon the use of private versions, we may support it in the future, though it adds certain complications to the underlying delta data structure for a number of reasons:

- A private version is not necessarily initialized from the previous private version. It may be initialized from another private version, from some released version anywhere in the version group, or even from an object in a different version group, or one that is not version controlled at all.
- Private versions are not necessarily frozen in sequential order.
- Frozen private versions are frequently deleted.

A reasonable approach would keep an *initialization link* when a private version is created, referencing the object used to initialize it. When a private version is frozen, a delta could be produced for it either relative to the latest released version on the branch, or where reasonable, relative to the first frozen version along the initialization chain.

The problem is that multiple arbitrary private versions may then be delta-compressed with respect to a frozen version. If that version is deleted, all versions delta-compressed with respect to it need to be found, and have their deltas recomputed.

Initialization links are particularly useful when a version is initialized from another frozen (private or released) version, and then immediately frozen without further modification[9]. There is no need to delta-compress the newly frozen version at all; it simply can be linked to its initializing object. In fact, linking together such versions is even more important when deltas are not used -- the space for a fully reconstituted version is saved.

7.4. Delaying Reconstitution

A user who retrieves a version may wish to retain a descriptor for it, storing it in a directory as a "hard link". As we noted in Section 7.2, this would ordinarily keep the reconstituted version around, even if it were not in active use. Using the LINK attribute described in section 6.1, it is easy to solve this problem.

Instead of storing a descriptor for a reconstituted version directly, a user can create a *Version Link* object, and store a descriptor for it in a directory instead. The version link object contains two components which identify the version -- a descriptor for a version group, and a string containing a version identifier (e.g. `"B3@5"`),

The version link object has a LINK attribute. When a user retrieves the descriptor for the version link object from the directory, the link is "followed" -- i.e. the type-specific LINK.`retrieve` method is called. It gets the version group descriptor and the version identifier out of the version link object, and (by calling the version group's DIRECTORY.`retrieve` method) retrieves the descriptor for the corresponding version, which is then returned to the user.

In effect, a version link object acts as a "semi-hard link" -- it contains a "hard link" to the version group, and then identifies the version through a version identifier. Indeed, the string placed in the version link object need not contain a version identifier -- it could contain an arbitrary version selector, which would then be re-evaluated each time the descriptor is retrieved, and the link is followed.

7.5. Avoiding Reconstitution

The file system used for DSEE [Leblang84] was extended to support reading a version controlled file without reconstituting it. Gypsy provides the ability to do the same, but uses the object-oriented extension mechanisms to keep the code out of the kernel.

When the Version Manager is asked to retrieve a delta-compressed version, it passes the version identifier and a descriptor for the version group to the DELTA.`retrieve` method defined for the object type. Ordinarily, this reconstitutes the object, and returns a descriptor for it, but this is not necessary. The method could return a descriptor for an object of a related type that masquerades as the requested object. It would define the same attributes as the original, but the methods would have different implementations. The masquerading object, like a version link object (Section 7.4), would contain a version group descriptor, and a version identifier, but would not have a LINK attribute.

Consider an object, like a file, which had a BYTE_STREAM attribute defined, through which the object could be read. To avoid reconstitution, when the corresponding DELTA.`retrieve` method was passed a version group descriptor and a version identifier as arguments, it would return an object of the masquerading type, which would simply contain the arguments passed.

Like the original object type, the masquerading type would also define a BYTE_STREAM attribute, through which the version could be read. When the masquerading object was opened (via BYTE_STEAM.`open`), and read (via BYTE_STEAM.`read`), the methods invoked would obtain the version group and version identifier from the masquerading object, and deliver bytes directly from the

[9] This might happen, for example, if the latest version on branch A were merged with the latest version on branch B (see Section 8), and the owners of both branches wanted to use the result as the basis for future development.

delta-compressed representation, without ever reconstituting the version.

8. Merging

Gypsy supports a standard three-way version merge -- that is, the changes made in going from version X to version Y are applied to version Z, yielding version Z'. Merging is not presently implemented; this section describes our current design.

To merge, the user specifies versions X, Y, and Z anywhere in the same version group, and the resulting Z' extends a branch also specified by the user. If the branch is unlocked, the new version is just placed at the end of the branch. If the branch is locked (by the user's current workspace, of course), a new private version is generated (with Z as its initializing link). The version identifiers of X, Y, and Z are stored as properties on Z', and can be obtained, for example, by a program that wishes to display the logical structure of the version group. Version X often is not specified -- it is then taken to be the nearest common ancestor of Y and Z.

8.1. Type-Specific Merging

Merge support for files is built into Gypsy; merging of other types of objects is type-specific. If the type supports deltas, then merging is done most efficiently on the deltas themselves. If there is a DELTA attribute, then it will contain a "merge" method; otherwise a "merge" method is found on a separate MERGE attribute.

When DELTA.`merge` is invoked, its arguments include version identifiers for X, Y and Z. In addition, a version identifier is provided which is to be used as the basis for delta-compression of Z'. That version identifier is chosen by the Version Manager based on its decision of how best to delta-compress Z'.

Ordinarily, Z' would be delta-compressed with respect to Z. But, if Z is not currently the latest released version on the branch where Z' is to be placed, the Version Manager might, for example, choose to use the latest released version instead. If it did not, a more complicated delta structure for released versions would be needed, similar to that described for private frozen versions in Section 7.3.

If deltas are not used, then the type-specific MERGE.`merge` method is passed X, Y, and Z, as retrieved from the version group, and produces Z', which the Version Manager simply stores as specified.

Arbitrary merge procedures can be called directly by a user; these would not only compute Z', but would release it to the version group, and store its antecedents as properties directly. This approach would be more suitable for tools like FileMerge [Adams86], that do 2-way, rather than 3-way merges, or that provide a fully interactive environment for merging.

8.2. Handling Conflicts

Merging can lead to conflicts [Reps88]. Consider a simple example: The only difference between X and Y is that the line "hello" has been inserted between lines 10 and 11. The only difference between X and Z is that the line "goodbye" has been inserted between lines 10 and 11. Should Z' contain just "hello" or just "goodbye", or both of them, and if so, in what order?

Various strategies have been used to resolve these conflicts.

- Conflict resolution rules may be provided.
- Data flow can be used [Reps88].
- Some choice is made, but a comment is also included indicating the nature of the conflict, or the description of the nature of the conflict is placed to a separate document.
- The user is interactively asked to decide.

We leave the choice to the user, who passes a (type-specific) procedure when the Merge is requested. The Version Manager then passes this procedure as a parameter to the type-specific "merge" method it invokes. That "merge" method calls the conflict resolution procedure each time it must resolve a conflict, passing type-specific arguments to it. The (type-specific) result indicates what "merge" should include at the site of the conflict.

9. Event Management

The Gypsy Event Manager provides support for monitoring changes to objects, and for triggering actions when an event occurs. Its functionality is similar to DSEE monitors [Leblang84b], however, Gypsy allows arbitrary programs to be executed as actions, and supports the inclusion of new types of events by individual type managers.

The user expresses interest in an event by making a *subscription*, consisting of a *target* object, a *condition*, and an *action*. For the action, the subscriber may choose either to receive notification, or to supply a program which will be executed when the event occurs.

An event may be a recurring event or a one-time event. A one-time event subscription is canceled when the event occurs. A recurring subscription remains in effect until it is explicitly canceled.

9.1. Object-Oriented Event Handling

The Event Manager posts a subscription to the target object's type manager which is responsible for associating it with the target object, and for monitoring the target for the occurrence of the event. A type manager agrees to monitor events by supplying an EVENT attribute with two methods: a "subscription" method which takes the target object and the condition, and which returns a *subscription*, and a "cancel" method which takes the target object and the subscription, and cancels it.

On a subscription, the Event Manager passes the *condition* on to the type manager without any interpretation; each type manager determines the sort of events it will monitor and interprets the condition accordingly. When an event is detected, the type manager signals the Event Manager, which then triggers the action associated with the subscription. The action is executed under the authority of the subscribing user.

Before triggering the action, the Event Manager first checks the authority list of the targetted object to ensure that the subscriber still has the right to access it. If not, the subscription is canceled and no action is triggered. The user is not notified of the cancelation -- notification of cancelation when the event occurs would be a violation of protection (see [Rotenberg74]). Type managers which perform additional access control based on internal state (such as the Version Manager) will need to retain the identity of the subscriber internally, and when an event occurs, check internally whether the subscriber still has the right to monitor it.

9.2. Version Management Events

The Version Manager supports subscriptions to events for version group objects. It supports a variety of conditions, which effectively allow monitoring of events such as:

- When a specified operation (e.g. release, destroy, change property) is taken on any version satisfying a specified predicate.
- When a specified operation (e.g. lock, unlock, release version) is taken on a specified branch.
- When as a result of some operation on a version group, a specified version no longer satisfies a specified predicate, or a new version is selected by a specified selector.

We expect that a user will most often monitor the unlocking of a branch (so that it can lock it), and the release of a new version (so that it can be used in building a configuration).

10. Configuration Management

The Configuration Manager supports construction of software products from versions of software components. The construction process is specified in a *configuration template* -- a text file which describes, among other things, the *components*, the *derived objects* (the objects which are to be built), and the *build steps* used to derive them. The configuration template is stored in a *configuration* object; as the derived objects are built, they are stored there as well.

- Each *component* is identified by a name, and specifies a directory pathname for locating its version group, and a version selector for choosing a specific version. Before it can be used in a build step, each specified component has to be bound to a particular version. The version specification can be reevaluated and the component rebound with each build operation. However, a user can explicitly direct the Configuration Manager to bind or rebind any component's pathname (i.e. as a hard link to a version group), and then optionally its version selector as well (to the version identifier obtained by evaluating the selector). Prebound pathnames or version selectors will not be reevaluated at build time, insulating a developer from changes made by others.
- Each *derived object* is identified by a name, and when built, is stored under that name in the configuration. A configuration, in fact, is a *customized directory*, and so ordinary directory pathnames can be used to name each derived object, as well as the components and the configuration template.
- Each *build step* specifies its inputs (components or derived objects), its outputs (derived objects), a *tool* used to produce them, and processing options for that tool. The actual commands to be used in the build step can be provided explicitly (as in Make [Feldman79]), however, Gypsy can generate them automatically from the inputs, outputs, and options by calling a method registered with the tool.

Configuration templates also can specify build dependencies (though Gypsy can also extract them automatically by calling a method registered with the tool), specific versions of tools to be used, environment variables to be set while building the configuration, and directory search paths for components.

Macros can be used to localize replicated information and to parameterize the configuration template. In particular, the version selectors of a number of components could use the same macro to select consistent versions, similar to Adele's single generic rule [Estublier88] or DSEE"s "configuration thread" [Leblang84b].

Gypsy configurations intentionally do not have a "consistency requirement". Differently named components can select different versions of the same version group, and can be used in building two different derived objects. This can be advantageous, especially when debugging [Schwanke88a].

The Configuration Manager also provides support for nested subconfigurations, including mechanisms to propagate dependency information, and to coordinate selection of component versions among the parent and the subconfigurations. For a detailed discussion of these issues, see [Schwanke88b].

10.1. Builds

When the Configuration Manager builds a derived object, it stores its derivation information in the configuration -- the bindings of the tool and inputs used

to build it, their timestamps, and the tool options. The Configuration Manager can avoid rebuilding a derived object if its derivation information is up to date. The user can also "bless" a derived object to avoid rebuilds.

We designed, but have not implemented, a Derived Object Manager that would maintain a system-wide cache of derived objects, and which would allow a Configuration Manager to reuse a derived object that was built in an entirely different configuration. Again, see [Schwanke88b] for details of these mechanisms.

Like Make, a *build* can invoke a number of build steps, with dependency information used to determine their sequencing. Builds are invoked explicitly, however, the event manager can be used to invoke a build on the release of a new version of some component.

10.2. Names, Locations, and Versions for Derived Objects

Because derived objects are stored in configurations, which are customized directory, they can be accessed using ordinary directory pathnames. When a derived object is rebuilt, it replaces the previous version of that derived object in the configuration -- there is no automatic versioning of derived objects.

These particular design decisions about the name, location and versioning of derived objects were intentionally chosen to present a simple user model that would be fairly close to that presented by Make. We did consider other design alternatives.

We considered requiring that a pathname for each derived object be provided. The derived object would be stored there instead of (or perhaps in addition to) the configuration. We felt, though, that it was best to mandate the configuration as the repository of derived objects. A user wanting to store a derived object elsewhere could do so explicitly, or, by using Configuration Events in conjunction with Event Management, could do so automatically each time the derived object was rebuilt.

We considered automatic versioning of derived objects -- either the configuration would maintain a version group for each derived object, or else the version group would reside elsewhere (named by a path as in the above alternative) with the configuration maintaining a link to the latest version. The derivation information would become additional properties for the version group.

However, we did not believe that maintaining versions of derived objects would generally be useful. This is especially true when components are private versions which are not frozen (which would typically be the case during heavy debugging), since the versions used to build the derived object might have been changed.

There is also no built-in automatic versioning of configurations. Configurations, like other objects, can be version controlled. Indeed, each version of a configuration can be thought of as the release of a system *baseline*. However, new baseline releases must be produced explicitly, though of course, the event manager can be used to trigger a new release on some event.

10.3. Hierarchical Version Control

When objects are hierarchically organized, a user may want to version control both the top level object as well as the sub-objects. For example, if a document is made up of a collections of chapters, it may be useful to version control each of the chapters separately, and also maintain versions of the document as a whole.

One method of coping with hierarchical versions is to *percolate* changes up the hierarchy; releasing a new version at one level automatically causes a release of a new version at each higher level [Zdonik86]. For example, releasing a new version of a chapter would automatically cause a new version of the entire document to be released.

Hierarchical version management in Gypsy can be provided, more cleanly, and with finer control, through the use of configurations. If chapters are individually version controlled, then a document can be represented as a (version-controlled) configuration with each chapter as a component.

A user can *explicitly* produce a new version of the document by making a new version of the configuration and binding its components. The specifications of each component in the configuration template selects a version for each chapter.

Not all of the chapters need to have their versions bound; the remaining chapters would be *dynamically bound* instead of *statically bound* [Kim87]. Each retrieval of an unbound chapter in a document would dynamically select the version to which its selector evaluates at that time.

11. Integration Problems

Using an object-oriented operating system as a basis for version management has provided a great deal of power and flexibility. Still, there were places where the object-oriented model was not powerful enough to support all our needs. This section discusses some of those problems and outlines some additional requirements.

11.1. Inheritance

Ideally, a program that expects to retrieve a descriptor for a file in order to read it should be willing to accept any type of object that has a BYTE_STREAM attribute defined. In fact, some programs simply check whether the object is a file. Such programs will, for example, have trouble if version-controlled files are not reconstituted when retrieved, but descriptors for masquerading objects, as described in Section 7.5, are returned instead.

In part, this is just a matter of educating program writers to identify objects by their attributes, rather than by their type.

It also points out, though, that attributes are a weak model of inheritance. If the operating system used inheritance instead of attributes, an object masquerading as a file would have a type that is a subtype of "file", and the question "are you a file?", would have been answered in the affirmative.

If the operating system did support multiple inheritance, we should have to consider whether a version group of objects of type T should be a subclass of type T [Zdonik86]. This is a tempting approach, for it lets the version group masquerade as the default selection.

In Gypsy, this is not necessary, since the LINK attribute provides this masquerading with even greater flexibility (Sections 6.1 and 7.4). The version group itself has the LINK attribute, providing the same masquerading as the multiple inheritance model. But in addition, with version link objects, retrieving a descriptor can even re-evaluate an arbitrary version selector.

11.2. Locating Reconstituted Versions

We noted in section 7.2 that removing a reconstituted version from the cache does not necessarily destroy it. Moreover, if a descriptor for it is retained elsewhere,[10] subsequent retrieval of the version will cause a new copy to be reconstituted, even if the old reconstituted copy still exists. That is unfortunate.

The problem could be solved if the operating system provided a mechanism whereby the owner of an object could somehow be notified when it held (in another object that it also owned) the only remaining descriptor for it.

With this scheme, descriptors for *all* of a version group's reconstituted objects (which are owned by Gypsy) could be stored in the reconstitution cache (which is also owned by Gypsy). When the Version Manager is notified that the cache holds the only remaining descriptor of the reconstituted object, it remembers to destroy it after a reasonable time if no attempt is made to subsequently retrieve it.

In summary, by adding an additional mechanism to the operating system, the cache could hold descriptors both for all reconstituted objects which have other outstanding descriptors, as well as for all recently used reconstituted objects.

11.3. Workspace-based Access Control

Oddly enough, even though the operating system has provided support for both descriptor-based and authority-based access control, we find that there are still some protection problems that are not easily solved.

In particular, we wanted to enforce workspace-based access control, and ensure that a private version could be accessed only when a user was attached to the workspace that had locked the corresponding branch or to another workspace that had been granted access (Section 4.4).

Unfortunately, a user who has been attached to a workspace, and retrieved a descriptor for a private version, may retain it even after detaching from the workspace. Currently, this cannot be prevented[11].

To implement the required access revocation, the Hydra system [Cohen75] proposed aliases, which are transparently interposed between the descriptor and the object it references (this would require some additional hardware support in our system, since Gypsy's descriptor-based addressing is implemented in hardware). The owner of the alias has the ability to break its connection with the object it references.

Using this mechanism, when asked to retrieve a private version, the Version Manager would create an alias for it which it would retain, and would hand out a descriptor for the alias. When the user detached from the workspace, the Version Manager would break the connections of all the aliases for those versions.

12. Concluding Remarks

A version of the Gypsy Version Manager was released in January 1988, and has been undergoing beta testing. A release with more complete functionality is due later this year.

There are a number of interesting directions that we would like to explore in the future:

Transactions are useful in software management; a user might enclose the release of a number of related versions within a transaction to ensure that other users see a consistent state. The operating system supports nested transactions, and interacts with type managers using a TRANSACTION attribute with methods for associating objects with transactions, resolving them, and participating in recovery. Each of the Gypsy type managers implement the TRANSACTION attribute, however, the Version Manager's implementation, in particular, could be redesigned to support greater concurrency.

The interaction of transactions with Event Management is particularly interesting. If an event occurs within a transaction, the action it triggers is delayed until the transaction is resolved. Otherwise, if the action has side effects and the transaction is aborted, there would be no way to recover from them. This means that applications which require notification of an event while a transaction is in progress cannot be supported. We would like to find a solution to this problem.

Gypsy will run on workstations connected by a local area network. We want to look at data structures for efficiently implementing distribution and replication of versions and version group fragments.

[10] We would hope that users who want to retain descriptors to versions would use version links (Section 7.4) rather than hard links, but that cannot be enforced.

[11] Though luckily, access cannot be retained indefinitely. To retain a descriptor, it must be stored in a persistent object. However, if the descriptor is retrieved by another job, the user will find that, because of the authority list associated with private versions, all access rights have been stripped away.

We are interested in delta representations of interesting object types, particularly trees and graphs, and user interfaces for displaying to users the differences between versions of them.

Finally, Gypsy is not built on top of an underlying database model as are some other recent systems [Dittrich86,Mahler88]. Nonetheless, it would be possible to present Gypsy as if it were -- with relations among versions, branches, workspaces, derived objects, and configurations appearing to be represented as database relations -- at least so far as queries are concerned.

Obtaining information by using a single query language rather than specialized query functions (e.g. list the versions in a branch, the branches locked by a workspace, etc.) would provide greater consistency, and the use of joins and generalized closures in the query language would provide additional power. Version predicates could be replaced by queries, though the query language may need to be crafted to provide support for expansion of default names (Section 6.1) and for integration with wildcard characters recognized by a shell (Section 5.1).

13. Acknowledgements

There are many individuals who have contributed to the design and implementation of Gypsy and its Version Manager. Peter Feiler and Walter Tichy were involved in the original system design. Their insight into the problems of software development was invaluable. Michael Rissman was instrumental in keeping the Gypsy user model simple and was responsible for suggesting that the naming and branching structure of version groups be decoupled. Marc Balcer, Julia Cohen, David Emery, and Elizabeth Wei participated in design, implementation, and testing. We are also grateful to Horst Mauersberg for arranging funding and initial support, to Tom Murphy and Pat Vroom for managing the development phase of the project, and to Clarice McDonald for managing the flow of documents.

Bibliography

[Adams86] Evan Adams et.al. SunPro: Engineering a Practical Program Development Environment. *Proceedings: Advanced Programming Environments*, Trondheim, Norway, June 1986. *Lecture Notes in Computer Science*, 244, Springer-Verlag.

[Bershad88] Brian N. Bershad & C. Brian Pinkerton. Watchdogs: Extending the UNIX File System. *Winter 1988 USENIX Technical Conference*, Dallas TX, February 1988.

[Cohen75] Ellis Cohen & David Jefferson. Protection in the Hydra Operating System. *Proceedings of the 5th Symposium on Operating System Principles*, Austin TX, November 1975. Operating Systems Review. 9(5).

[Curry84] Gael A. Curry & Robert M. Ayers. Experience with Traits in the Xerox Star Workstation. *IEEE Transaction on Software Engineering*, SE-10(5), September 1984.

[Belkhatir86] N. Belkhatir & J. Estublier. Protection and Cooperation in a Software Engineering Environment. *Proceedings: Advanced Programming Environments*, Trondheim, Norway, June 1986. *Lecture Notes in Computer Science*, 244, Springer-Verlag.

[Dittrich86] K. Dittrich, W. Gothard, P.C. Lockemann. Damokles: A data base system for software engineering environments. *Proceedings: Advanced Programming Environments*, Trondheim, Norway, June 1986. *Lecture Notes in Computer Science*, 244, Springer-Verlag.

[Estublier88] Jacky Estublier. Configuration Management: The Notion and the Tools. *Internation Workshop on Software Version and Configuration Control*, Grassau, FRG, January 1988.

[Feldman79] Stuart I. Feldman, Make -- A Program for Maintaining Computer Programs. *Software Practice and Engineering*, April 1979.

[Goldberg83] A. Goldberg & D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, May 1983.

[Kaiser87] Gail E. Kaiser & Peter H. Feiler. Intelligent Assistance without Artificial Intelligence. *Proceedings of the 9th International Conference on Software Engineering*, IEEE, February 1987.

[Kim87] Won Kim et.al. Composite Object Support in an Object-Oriented Database. OOPSLA 87. Orlando, Florida, October, 1987.

[Leblang84] David B. Leblang & Robert P. Chase, Jr. Computer-Aided Software Engineering in a Distributed Workstation Environment, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh PA, April 1984. SIGPLAN Notices, 9(3), May 1984.

[Leblang84b] David B. Leblang & Gordon D. McLean, Jr. Configuration Management for Large-Scale Software Development Efforts. In *Workshop on Software Engineering Environments for Programming-in-the-Large.* June 1984.

[Mahler88] Axel Mahler & Andreas Lampen. **shape** -- a software configuration management tool. *Internation Workshop on Software Version and Configuration Control*, Grassau, FRG, January 1988.

[Narayanaswamy88] K. Narayanaswamy. Version Control in the Common Lisp Framework. *Internation Workshop on Software Version and Configuration Control*, Grassau, FRG, January 1988.

[Pollack81] Fred J. Pollack, et.al. The iMAX432 Object Filing System. *Proceedings of the 8th Symposium on Operating System Principles*, Pacific Grove, CA, Dec 1981. Operating Systems Review 15(5).

[Reps88] Thomas Reps et.al. Support for Integrating Program Variants in an Environment for Programming in the Large. *Internation Workshop on Software Version and Configuration Control*, Grassau, FRG, January 1988.

[Rochkind75] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4), December 1975.

[Rotenberg74] Leo J. Rotenberg. Making Computers Keep Secrets. Ph.D. Thesis. MAC TR-115. MIT. February 1974.

[Saltzer75] Jerome H. Saltzer & Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), September 1975.

[Schwanke88a] Robert W. Schwanke & Gail E. Kaiser. Living with Inconsistency in Large Systems. *Internation Workshop on Software Version and Configuration Control*, Grassau, FRG, January 1988.

[Schwanke88b] Robert W> Schwanke, et.al. Configuration Management in Gypsy. Siemens RTL TR-88-Schwanke, April 1988.

[Tichy85] Walter F. Tichy. RCS -- A system for version control. *Software -- Practice and Experience*, 15(7), July 1985.

[Tichy88] Walter F. Tichy. Tools for Software Configuration Management. *Internation Workshop on Software Version and Configuration Control*, Grassau, FRG, January 1988.

[Wulf81] William A. Wulf, Roy Levin, Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[Zdonik86] Stanley B. Zdonik, Version Management in an Object-Oriented Database. *Proceedings: Advanced Programming Environments*, Trondheim, Norway, June 1986. *Lecture Notes in Computer Science*, 244, Springer-Verlag.