At Harlequin created on September 29, 1994 and last modified at Harlequin on November 11, 1994.

Source Information Database

Scott McKay & John Dunning

1. Introduction

The purpose of this paper is to describe the source information database, that is, the database that models all program "source". By program "source", we mean actual source code, requirements and specifications, user documentation, test suites and examples, bug tracking information, project management entities (schedules and task lists), and so forth.

We do not propose that the source information database serve the entire role of a full-featured, unified source version control and system configuration management tool — at least not initially. It must, however, provide a basic set of facilities that support these tasks.

Note that the source database does not have any direct linguistic support for program sources. This support is provided by other tools, and is recorded in the derived information database. This is all covered in the white paper that describes the derived database.

It is the intention that the source information database be implemented using a true, object-oriented database substrate

2. Basic Design Goals

The most basic design goals are:

- Provide basic source version control functionality, with granularity at both the "section" (i.e., language definition) level and the file level. That is, single language definitions are first-class objects as well as files.
- Support distributed sites, including intermittently connected portable or home computers.
- Provide basic system configuration management functionality.
- All important entities, such as sections, files, and projects, must support multiple branches and versions. Access to branches and versions should be simple and transparent.
- Support "hypertext links" between sections of arbitrary types (for example, source code might point to its specification).
- Allow various kinds of data, include text, graphics, audio, and so forth.
- Have a client-server architecture, with things like locks and consistency being provided by the underlying database substrate.

The version control functionality has the following goals:

- Support separate lines of development via "branches"; this provides one way of managing group development. These branches can be split or joined, and named. Creation of a new "workspace" (by creating a private branch) should be very simple.
- Have basic support for conflict resolution ("the merging problem").
- Have a fine granularity — the "section" — that roughly corresponds to one definition in a programming language. The traditional notion of a file is built up from an ordered set of sections.
- Support change histories that record the author, date of change, and the purpose of the change.
- Allow importing existing source information (which may exist in "flat" files or source control systems such as RCS) into the database.
- Allow exporting source information into some "interchange" format (such as flat files).

The system configuration management functionality has the following goals:

- Support grouping of files into projects.
- Support "snapshotting" of particular configurations of source files and projects.

Note that not all of these goals are targetted for the initial release of DylanWorks. For example, the support for distributed may be "layered" functionality.

3. The Source Database Schema

The source database for a project itself resides in a single database file that we will call the source information database. This database contains all of the branches of all of the files in the project.

Other information about the project is stored in its derived information database. There are pointers from the derived database to the source database, but no pointers from the source database to the derived database.

3.1 `Versions`

A *version* is a particular state of a file, a project, or the contents of a section. Note that a version is not a first-class object in the database; it is just a useful abstraction.

Versions of an object are numbered with a version number that increases by 1 each time a new version of the object is saved out.

Any object that can be versioned has the following attributes in addition to its other attributes: author, creation date, and modification comments.

The term *section version* includes the text of the section. This text may be represented as a difference set from another version of the same section.

The term *file version* refers to a set of sections for that version of the file. It also has other attributes, such as a “do-not-delete” flag.

The term *project version* refers to a set of files in that version of the project.

3.2 The `branch` entity

A *branch* is a sequence of file versions, and is identified by a branch name. It can be thought of as a distinct line of descent for a file or project (set of files). Branches can be newly created or *split* from an existing branch of a file or project.

One important piece of functionality on a branch is determining its “heritage”, that is, when it was created, what its ancestors were, what files the branch included (if it is a branch of a project).

The attributes of a branch are its name and its “heritage”.

The only operation besides creating it and showing its heritage is to rename it. Otherwise, operations are done on files and projects.

3.2.1 Branch Search Lists

A branch search list specifies one or more branches that will be searched in order to find an entity in the source database. If the entity is not found in the “selected” branch, the branches named in the branch search list are search, in order, until the entity is found.

Branch search lists are typically specified in projects and in user profiles.

3.3 The `file` entity

A *file* is one version of a single branch of descent. It consists of an ordered set of sections.

Note that most DylanWorks tools will operate at the granularity of a section. We choose to retain the notion of a file because it provides a convenient handle for certain kinds of things, not the least of which is a sense of familiarity for most people who will use DylanWorks.

It is occasionally useful to refer to all of the versions of a file, that is, the thing that contains all of the branches of that file. We use the term *metafile* to refer to this. Using this terminology, a file is a single version from one branch of its owning metafile.

A file is a versioned entity, in that each file in a branch has a monotonically increasing version identifier. Each version of a file has its own set of sections. The other attributes of a version of a file are its author, creation date, and modification comments.

The Dylan Pathname Library will provide for naming these file with Dylan pathnames.

Files can be checked in or out, locked, unlocked, split, joined, sealed, opened, or deleted. It is also possible to show or change the attributes of a file, and to show the history of a file.

Note that this definition of a file is intentionally chosen to mimic the ubiquitous flat-file metaphor, while at the same time allowing for hypertext structures. Among the reasons that we are retaining the notion of a file (albeit implemented differently) is that it provides a way of capturing the intent of the original author(s) with respect to grouping of language definitions, when that grouping cannot be derived any other way from the program text. A file also give a useful handle for reducing the complexity of system configuration management. It also provides a convenient mapping into existing code management systems.

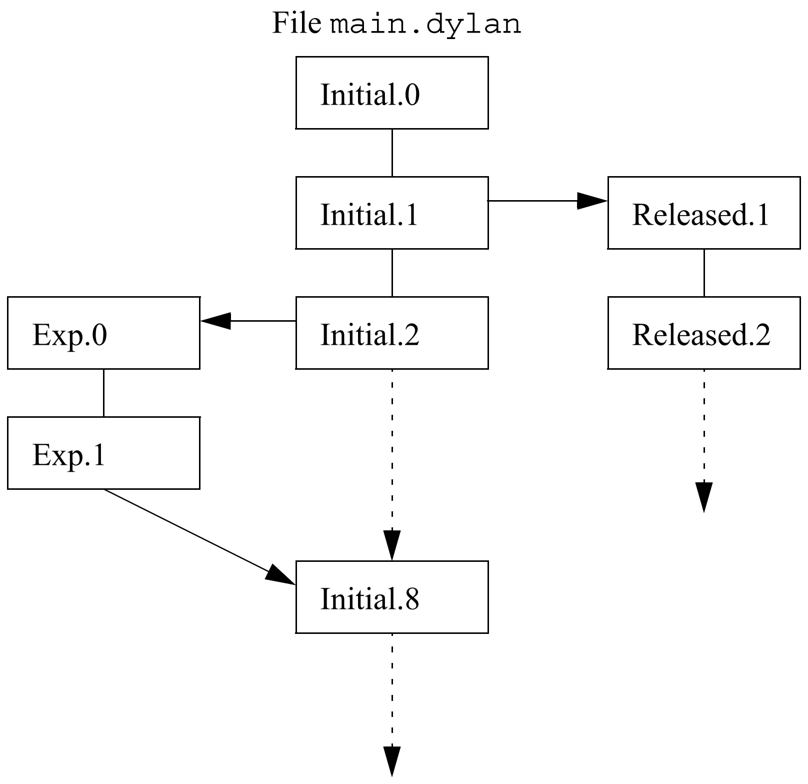

Figure 1. A file with three branches.

3.4 The `directory` entity

A directory is a way of constructing a hierarchy of names, just like in a traditional file system. A directory can contain either metafiles or other directories. Directories are used for grouping related things together, just as they are in traditional file systems.

A directory's attributes are its pathname, the author who created it, its creation date, and the set of directories and metafiles it contains.

Directories can be created, deleted, and renamed. Users can also specify that they wish to use a default directory.

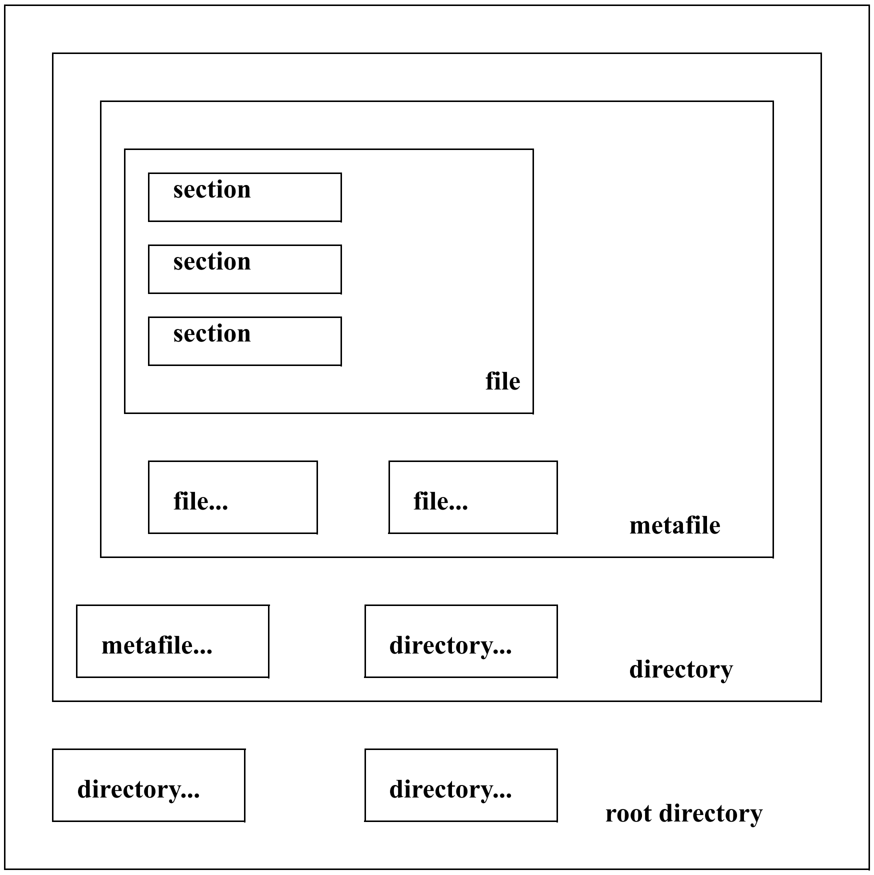

Figure 2. The structure of a source database.

3.5 The `section` entity

A *section* is a simply an atomic sequence of characters. The basic section class contains plain ASCII characters, but there may be subclasses that contain Unicode characters, encodings of images, and so forth.

A section belongs to a single file, but note that a section may belong to many versions of a file.

Each section has a unique “handle” that may or may not be a meaningful, human-readable name. In DylanWorks, the derived information database keeps a mapping from Dylan module variable names to section handles. (Note that this means that a source code section may contain one or more language definitions.)

Sections are, in effect, versioned entities, but since sections are mostly “invisible”, the versioning may be implemented by simply choosing a new handle name for a new version of a section. Whenever a new version of a section is saved out, the version number of its owning file is also incremented. The most recent (“newest”) version of a file consists of the most recent versions of all of its sections.

Sections can be checked in or out, locked, or unlocked. It is also possible to show the history of a section.

The attributes of a section include:

- The “text” of the section (which may be things other than plain ASCII text, such as graphics, audio, encrypted text, and so forth).
- A “property list” that is used to encode such things as the current status of the section (such as “completed”, “broken”, “experimental”).

The other attributes of a version of a section are its author, creation date, and modification comments.

It is intended that there be subclasses of sections that are used to store certain kinds of data (for example, source code sections, documentation sections, bug mail sections, and so on). These subclasses will record other information; for example, a source code section will have hypertext links to its specification, user documentation, bug reports, test suites, and so on.

The definitions in a source file are typically represented as a set of sections, with one section being used to represent each source definition. This “sectionization” is done either by having a user explicitly create a new section while editing source code, or by a module of the editor that implements a simple language-specific parser that breaks the source code into sections. The automatic sectionization process might not be completely accurate, but since the derived information database keeps an accurate mapping from language definition names to section handles, any inaccuracy is not harmful.

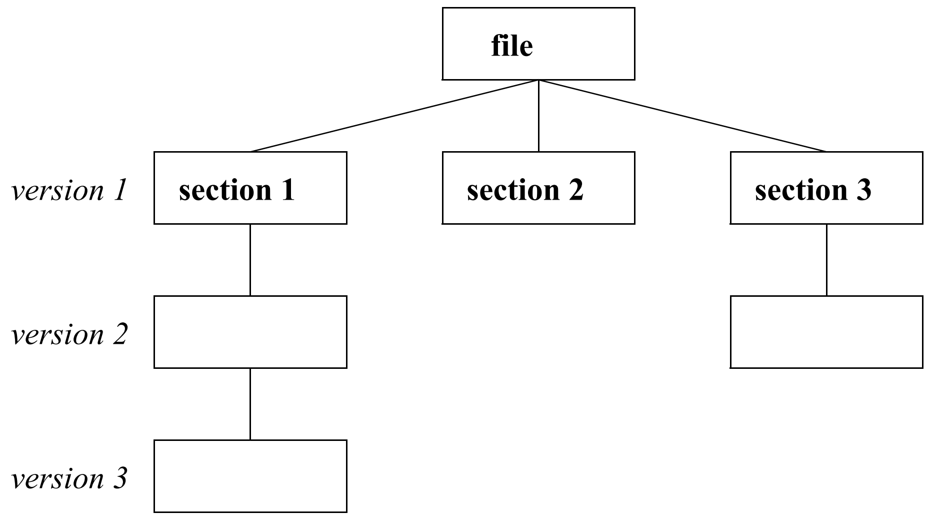

Figure 3. Three versions of a file and their sections

3.6 The `snapshot` entity

A *snapshot* is an object that stores a set of file or project branches and versions. It allows users to identify all of the files or projects in the snapshot at a given point in time. The snapshot will presumably comprise some meaningful state of the project's evolution.

The attributes of a snapshot include its name, author, creation date, and the set of file and project versions.

Snapshots are used by the system configuration management tools. For instance, compiling a project creates a snapshot of the project so as to capture the state of the project at the time it was compiled. This serves to "journal" the project so that its state can be reproduced sometime in the future.

3.7 The `project` entity

A *project* consists of a set of files (or other projects), a set of dependencies between the files, and a set of operations used to operate over those files (such as compilation and loading). The other attributes of a version of a project are its author, creation date, and modification comments.

A project can be checked in or out, locked, unlocked, split, joined, sealed, opened, or deleted. Doing these operations on a project is roughly equivalent to doing these operations on each of the files in the project.

Projects are versioned entities.

We envision a Dylan **define project** form that will be used to create a project.

The System Configuration Management paper describe the role of projects in more detail.

3.7.1 The Dylan `library` entity

A Dylan library entity is a kind of project that corresponds to a Dylan **define library** form.

Blah blah blah...

3.7.2 The Dylan `module` entity

A Dylan module entity corresponds to a Dylan **define module** form.

Blah blah blah... Does a module equal a "file" in DylanWorks?

3.8 Simple Entities

These are definitions for the simple entity types we use above:

- A *date* is a 32-bit integer representing a date and time.

- A *modification comment* is a text string.
- A *branch name* is a text string. It is case-preserving, but not case-sensitive. There is a single flat namespace for branch names within a single source database.
- A *snapshot name* is a text string. It is case-preserving, but not case-sensitive. There is a single flat namespace for snapshot names within a single source database.

3.9 Management entities

The source database will have types of sections that describe management entities such as:

- Schedules
- To-do lists
- Resources (personnel, machines, etc.)

3.10 Documentation entities

The source database will have types of sections that describe documentation entities such as:

- Specifications
- User documentation
- “One-liner” documentation, if it is not stored with source code entities
- Implementor documentation

Note that these entities will probably not contain the documentation, but will more likely describe how to extract this information from another source (such as a Frame document).

3.11 Other entities

The source database will have types of sections that describe documentation entities such as:

- Test suites
- Examples
- Bugs and bug reports (mail messages)

4. Operations on the Source Database

4.1 Create

Creating a file is to add a new file to the source database; do not confuse this with creating a new version of an existing file. Doing this entails naming the new file.

Creating a directory adds a new directory to the source database’s directory tree.

Creating a section is to add a new section to an existing file.

Creating a snapshot is to give a name to a set of files or projects at some point in time.

4.2 Delete

Deleting a file removes that branch from the source database. Deleting one version of a file merges it into the next version of the file, and then removes it from the database.

4.3 Import

It is possible to import data from other version control systems, such as RCS. Doing so entails creating new files in the source database, and importing the contents and revision history of the external file.

A flat file can be imported by either creating a new file in the database, using the flat file as the initial contents, or by checking in the flat file.

When importing a set of flat files that come from different directories, the directory structure of the file system will be mirrored in the directory structure of the source database.

4.4 Export

Exporting a file from the database simply creates the appropriate flat file. This is like checking out a file, except that no lock is taken.

When exporting a set of flat files that reside in different directories, the directory structure of the source database will be mirrored in the directory structure of the file system into which the exported files are written.

4.5 Check in

Checking in a file entails taking the contents of a (previously checked-out) file, and creating a new version of the file in the database that contains the modifications given by the file being checked in. Checking in a set of files obeys the same directory structure-preserving properties as importing.

Checking in a section is equivalent to checking in the section’s owning file with a new version of the section.

Checking in a project or a snapshot means to split all the files in the project or snapshot.

Note that DylanWorks will provide a short-cut to checking out and checking in files, in that many operations (such as compilation and editing) can be done directly on files stored in the database.

4.6 Check out

Checking out a file entails creating a flat file and copying the contents of the file from the database into the flat file. Sometimes the database file will be locked if the user anticipates changing the checked-out file.

Files, projects, and snapshots may all be checked out. Checking out a project or a snapshot means to check-out all the files in the project or snapshot. Checking out a set of files obeys the same directory structure-preserving properties as exporting.

Checking out a section simply creates a file with the section in it. This section can be checked back into its home file at a later time.

4.7 Lock

Locking protects an object while a user is using it. Objects can be locked for write by a single user, and locked for read by multiple users. If any user holds a read lock on an object, this means that no other user can take a write lock on the same object. For the most part, this sort of locking is maintained by the database substrate and is invisible to users.

Operations such as checking in or checking out a file in cause the file to be locked for the duration of the operation.

4.8 Unlock

Unlocking an object simply release the lock on the object.

4.9 Split

To split an object is to create a new branch (not version) of an object. The new branch is created by effectively copying the contents of the object being split. The new branch of each file starts at version 1, and remembers its “heritage”.

Files, projects, and snapshots may be split. Splitting a project or a snapshot means to split all the files in the project or snapshot.

4.10 Join

Joining is complementary to splitting. Joining a source branch with a target branch applies the changes in the source branch to the target branch, creating a new version of the target.

Files, projects, and snapshots may be joined. Joining a project or a snapshot means to split all the files in the project or snapshot.

4.11 Seal

To seal an object is to prevent further changes to it. (Some systems call this “capping”.)

Files, projects, and snapshots can be sealed. Sealing a project or a snapshot means to split all the files in the project or snapshot.

4.12 Open

Opening (sometimes called uncapping) an object undoes a previous sealing operation on that object.

4.13 Rename

Users can rename branches and snapshots. This simply changes their name string.

4.14 Show History

The show history operation can be used to examine the changes to a versioned object. The operation allows filtering by date, author, version number, and comment. You can show the history of a section, a file, or a project.

It is envisioned that browsers will provide user interface to this functionality, and that the source database system will simply provide a way to get this information.

4.15 Compare Versions

Two versions of an object can be compared to see how they are different. The objects may be two file versions, two sections, or two snapshots.

5. Future Work

There are a number of issues that we will probably not address in the initial release. The initial release, as we stated above, will provide the minimum useful functionality; other functionality could be provided either by extending our own system, or replacing it by something that models an off-the-shelf system such as Atria’s ClearCase. Functionality we do not plan to provide includes the following, even though this is all admittedly valuable:

- Highly efficient storage representation. The initial release will probably not store compressed “version deltas”, unless excessive size becomes a problem.
- Fully automatic merging for conflict resolution.
- Events and triggers.
- Access control lists and other types of security.

Other features to consider that will probably be omitted in the first release are:

- Reference directories (i.e., a directory into which a file is automatically checked out as the file is checked into the database).
- Exporting files directly into other version control systems such as RCS.
- “Mounting” the database as a file system so that it can be transparently accessed with other utilities.

6. Open Issues

The most significant issue is, why are we doing our own source control system at all, given the presence of excellent products such as Atria’s ClearCase. The main reason is that ClearCase still doesn’t properly support distributed sites, and more importantly, it does not support granularity at a fine enough level (i.e., the definition). We anticipate that, before long, there will be a commercially available product that meets our needs, and that should use it when it becomes available. Therefore, we are strongly motivated towards making the design of our own source control system as simple as possible.

The quest for simplicity means that there are some things we are not going to do, at least not until we decide that it might be worth pursuing the whole source control stuff further. For example, having “nested” sections and “nested” files is tempting, but we don’t plan to do this in the initial release. Having more general file-like things where the sections are generated on the fly (for example, the “callers of FOO” file-oid) is also tempting, but requires support for some kind of description language in the database; we don’t plan to this in the initial release either.

Note that a file serves double duty in this scheme. In an ideal world, its principal duty is to serve as an object that captures an author’s intent. However, in the “real” world, compilation is driven off of files so that things get compiled in the right order. Is there a way we can separate these two roles? Can the system configuration tools drive the compiler in the “correct” order without losing the author’s intent? (For example, we might analyze each section as it is saved in order to compute its ordering dependencies, and the system configuration tool can use that information. If we do this, what granularity does the SCM do journalling at?)