ArX

16th November 2005

Credits

ArX has been cooperatively developed, and has contributions from many people and organizations. A hopefully complete list: Pau Aliagas, David Allouche, Tim Barbour, Stig Brautaset, Jon Buller, Junio C Hamano, Environment Canada - Meteorological Service of Canada, Mike Coleman, Robert Collins, Don Dayley, Alexander Deruwe, Federico Di Gregorio, Nicholas Dille, Paul Eggert, John Ellson, Robin Farine, Lele Gaifax, Karel Gardas, Johnathan Geisler, Jonathan Geisler, Chris Gray, Jan Harkes, Isamu Hasegawa, Joey Hess, Mikael Hillerstrom, David Kantowitz, Walter Landry, Tom Lord, Andrew Morton, Frank Murphy, Steve Murphy, Gergely Nagy, Matthias Neeracher, Daniele Nicolodi, Scott Parish, Chris Paulson-Ellis, Ulrich Pfeifer, Marc Recht, The Regents of the University of California, Kevin Smith, Richard Stallman, Bruce Stephens, Robert W. Anderson, Bryan W. Headley, Martin Waitz, Colin Walters.

In addition, ArX makes use of some wonderful tools from the FSF (www.gnu.org) and four excellent libraries: Boost (www.boost.org), Loki (http://sourceforge.net/projects/loki-lib/), Brian Gladman's SHA implementation, and Graydon Hoare's xdelta implementation. The code in those libraries requires the following acknowledgements:

```
/* Copyright (c) 2000-2002
 * CrystalClear Software, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  CrystalClear Software makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.  */
 * Copyright (c) 1998-2002
 * Dr John Maddock
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Dr John Maddock makes no representations
 * about the suitability of this software for any purpose.
 * It is provided "as is" without express or implied warranty.
```

```
// The Loki Library
// Copyright (c) 2001 by Andrei Alexandrescu
// This code accompanies the book:
// Alexandrescu, Andrei. "Modern C++ Design: Generic Programming and Design
//     Patterns Applied". Copyright (c) 2001. Addison-Wesley.
// Permission to use, copy, modify, distribute and sell this software for any
//     purpose is hereby granted without fee, provided that the above copyright
//     notice appear in all copies and that both that copyright notice and this
//     permission notice appear in supporting documentation.
// The author or Addison-Wesley Longman make no representations about the
//     suitability of this software for any purpose. It is provided "as is"
//     without express or implied warranty.
```

Contents

Chapter 1

Introduction

ArX is a version control system that enables you to do many things that seem difficult or painful with current systems. Suppose you are creating something, be it a program, a document, or even graphics. As you make modifications to the work, you can save the different revisions into an archive as you go along. Then, if you decide that something you deleted is still useful, you can get that old work back. Sometimes it is just the difference between two revisions that is interesting. ArX also makes it easy to get just those differences.

As the work becomes larger and more complicated, it spreads into different files. Sometimes you make a number of related changes to a number of files, and you want all of these changes to be committed at the same time. In particular, some of these changes may depend upon each other. ArX supports whole-tree commits, which ensure that all of those changes are grouped together.

As the project matures, the logical structure changes, so you move files and directories around. You find it convenient to use symlinks and permissions. ArX stores all of that information, allowing you to get back exactly what you put in. Sometimes, you start working on a change that you may not be completely sure whether it will end up in the final creation. ArX makes it easy to create a branch of your creation that lives in parallel with the main line of development. Once the work on that branch is done, ArX makes it easy to integrate those changes back into the main line of development. Or you can just continue to work on the branch and completely forget about the "main" line.

Finally, you want to release your work upon the world. ArX supports ways to package up your creation into simple tarballs. People admire your work, and want to help out. This is where ArX's strengths really shine. You can publish your archive so that other people can watch your development, trying out new elements as you create them. You can use a variety of ordinary servers, including an ordinary web server, a web server with webDAV, an ftp server, or an sftp server for secure access. You can also digitally sign the archive to reduce the risk of someone compromising the code and inserted hidden bugs.

As time goes on, some of the testers become developers, sending in small patches to improve this or that part. They can work in isolation, creating patches that they send to you. You continue to work, and ArX makes it easy for independent developers to keep their tree up to date. They, too, can publish an archive, and ArX makes it easy to integrate patches from them. You can either take everything that they do, or you can selectively apply patches from among the ones they offer. As before, you can easily create branches to

try out patches from many different sources, and only integrate those that pan out.

In time, you may tire of your creation, and some of your contributors may become more prolific than yourself. ArX makes it easy for anyone to mirror your archive, and anyone can create their own branches. Anyone can become a new maintainer. A new person, or a new group, may come to hold sway over the future of the creation. Their own archives will become the centers of eagerly awaited patches, while your own fades into history.

Chapter 2

Installation and Versioning

2.1 Building ArX

To build the code you will need a decent C++ compiler. A recent version of gcc (>=3.2) is recommended. The code uses autoconf, so a minimal shell is needed. In addition, you must have Python >=2 (just used for building) and a working gnome-vfs2 install. Finally, ArX uses GNU diff, patch and tar. On Windows, you can get these with Cygwin. See INSTALL.CYGWIN for more details. On Mac OS X, you can use either pkgsrc[1] or Fink[2]. Detailed installation instructions can be found in INSTALL.GENERIC.

Once you have it installed and in your path, you can invoke it

```
$ arx
```

and it will give some output

```
Invoke a sub-command of arx.
usage: arx command [options] [arguments]
All commands take the following options:
 -h -H --help                  print a help message specific to that command
 --silent                      no output
 --quiet                       only output errors
 --default-output              default output
 --report                      slightly verbose output
 --verbose                     maximal output
 --                            mark the end of options
The -- option is useful for explicitly ending the list of options. This
is useful if a filename, for example, might be mistaken for an option.
In addition, you can specify the following options instead of a command.
 -V --version                  print version info
 -h --help                     display this help
 -H --help-commands            display a list of subcommands
```

[1] http://www.netbsd.org/Documentation/software/packages.html
[2] http://fink.sourceforge.net

In general, to invoke an ArX command, you type `arx` followed by the command, then followed by any options, and finally any arguments to the command.

2.2 Versioning

You can find out what version of ArX you are running with the `-V` option:

```
$ arx -V
ArX 2.2.2
Built 00:25:42 Apr 28 2005 with gpg support
Copyright 2001-2005 by various contributors.  See CREDITS for details.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Report bugs to <arx-users@nongnu.org>.
```

The version has three parts, a major, minor, and revision number. Two versions that differ only in revision number (e.g. 2.0.12 and 2.0.15) should be mostly compatible. Commands may be added, modified, or removed, but the disk format will be the same. A minor version change means that the format for anything except an archive may have changed. For example, you may have to delete and re-get a project tree. A major version change means that the archive format has changed, perhaps requiring you to convert archives.

Chapter 3

Setup

3.1 IDs

The first thing you should do is come up with an ID for yourself. ArX uses it to identify who committed a change to an archive, and who is holding locks on an archive. You can use your name or a pseudonym, as in

```
$ arx param id ‘‘Don Quixote de la Mancha’’
```

The argument is quoted because there are spaces in the name. To see what your ID is, you can use `param id` without arguments

```
$ arx param id
Don Quixote de la Mancha
```

3.2 Archives

Now you need a place to store all of the revisions you are going to be making. This is called an *archive*, and you create one with the `make-archive` command. You need to know two things before you create the archive: what to call it and where it will be. You can name an archive whatever you like, as long as it does not have a slash "/" or colon ":" inside it. The usual choice follows a syntax that looks like an email address followed by a further delineation. For example

```
dcoyote@example.org--archive
```

The advantage of using an email address is that it is already unique, so you won't have clashes with another person. You will probably find yourself creating different archives for different purposes, so it is wise to put some sort of qualifier at the end of the name of the archive. For example, if you want to have an archive for your work at Yoyodyne Inc. and another for your spare time, you might choose archive names such as

```
dcoyote@example.org--yoyodyne
dcoyote@example.org--freetime
```

Now that you have a name for your archive, you need a place to put it. Any spare directory with a fair amount of space will do. If you have decided to put the archive in the directory `archive`, then you make the archive with `make-archive`

```
$ arx make-archive dcoyote@example.org--archive archive
```

Note that you should **never** have to look inside that `archive` directory.

The last thing that you should do is set up a default archive. You just use the symbolic name of the archive

```
$ arx param default-archive dcoyote@foobar.org--archive
```

You are now set up for basic revision control.

Chapter 4

Basic Revision Control

4.1 The First Revision

ArX works on entire project trees, so everything has to be stored within a directory. This differs from some simpler revision control systems such as RCS which can operate on just one file. So we need to create a directory to store our files. We will illustrate with the simplest shell script, Hello world. We create the directory

```
$ mkdir hello
```

and then create the program

```
$ cd hello
$ echo "echo Hello, World" > Hello
```

Now we are going to store this masterpiece in ArX. We first have to initialize the tree, letting ArX know that we are creating a new project. We will call the project "hello".

```
$ arx init hello
```

This creates an `_arx` sub-directory in the current directory. You should never need to look at things in the `_arx` directory. It also automatically recursively adds all of the paths (files and directories) in the directory to the list of paths that will get stored in the archive.

You can now store the first revision in your archive by running

```
$ arx commit -s ‘‘First revision’’
```

If later you decide that you want to get back this initial revision, you use `get`

```
$ arx get hello,0
```

This is the initial revision, so it has ‘‘`,0`’’ appended to the end of the revision. Later revisions will have ‘‘`,1`”, ‘‘`,2`”, etc. appended.

4.2 Further Revisions

Now suppose that you have made some modifications to your project. Commiting the changes is just

```
$ arx commit -s ‘‘Fixed foo to do bar instead of baz’’
```

You can continue this simple scheme ad infinitum as long as you don't need to add, delete, or move files.

4.3 More complicated changes

Sooner rather than later, you will want to add more files. ArX requires you to explicitly notify ArX every time you add a file. You do this with the `add` command. For example, if you created a file named "Goodbye", you can add it by typing

```
$ arx add Goodbye
```

This works on directories as well, although not recursively by default. If you do not explicitly add a file, then ArX will not store it or any modifications into the archive. If you do not add a directory, any changes that occur in that directory will not be recorded.

Similarly, if you later decide that you don't need "Goodbye" anymore, you can delete it with the command

```
$ arx rm Goodbye
```

`arx rm` supports most of the same semantics as rm(1), so you can recursively and interactively delete files and directories. Finally, you can move files

```
$ arx mv Goodbye Goodbye.sh
```

`arx mv` supports most of the semantics of plain old mv(1), so you can move a number of files into a subdirectory

```
$ arx mv foo bar baz bat/
```

If you forget to use the ArX functions to delete and move files and directories, ArX will not let you commit. You can use the `tree-lint` to see what kinds of problems might arise during commit.

```
$ arx tree-lint
```

Some of the things that `tree-lint` complains about are only warnings that will not stop a commit. For example, if you forget to `add` a file. In general, if you have been doing a lot of modifications to the tree, it is wise to run `tree-lint` before committing.

If you have made so many changes over such a long time that you have forgotten exactly what you have done, `then`

```
$ arx diff
```

will tell you what paths have changed since the last time you committed.

4.4 Reviewing your work

You can get a terse listing of the revisions you have committed with

```
arx log
```

Alternately,

```
arx log --formatted
```

will give a more detailed picture. These commands look in the project tree for the information. If you are not in the project tree, then

```
arx log --remote
```

will instead query the archive. Given this listing of revisions, you can get a particular revision (e.g. revision 12) with

```
arx get hello,12
```

This puts revision 12 into the directory `hello.12`. Note that this is different from CVS, because you didn't have to explicitly tag a revision in order to get a particular snapshot of the tree. Every revision is akin to a snapshot.

4.5 Working with an existing project

Suppose your trusty sidekick Sancho Panza has set up an archive at

```
ftp://ftp.example.org/~spanza/archive/
```

You can register that archive with the archives command

```
arx archives -a ftp://ftp.example.org/~spanza/archive/
```

ArX will retrieve the name of the archive from the archive itself. If Sancho Panza named the archive `spanza@example.org` when he created it, then running `archives` without arguments should give you output like

```
$ arx archives
spanza@example.org
    ftp://ftp.example.org/~spanza/archive
```

However, you don't need to explicitly register the archive. ArX will do it for you whenever you access the archive (e.g. when using `browse` or `get`). So to see what is there, you can run

```
arx browse ftp://ftp.example.org/~spanza/archive/
```

The trailing slash "/" is important. ArX assumes that everything past the last slash is a branch name, so without the trailing slash ArX would be looking for the archive branch in an archive located at ftp://ftp.example.org/~spanza/.

Now that the archive is registered, you can use the archive name instead of URL's. For example, to get the windmill project from that archive, you can use

```
arx get spanza@example.org/windmill windmill
```

But using the full URL will always work

```
arx get ftp://ftp.example.org/~spanza/archive/windmill windmill
```

Once you have this project, you can keep up to date with any changes to the project with

```
arx merge --dir windmill
```

The `merge` command is analogous to the update command in CVS, although `merge` is much more powerful. You can set up ArX to pop up a graphical merge tool in case of conflicts. See Appendix C for details.

Chapter 5

Advanced ArX Concepts

The preceding chapters gave a basic introduction to working with ArX. However, some of the things that make ArX so useful necessarily become somewhat specialized. So it has been broken down into separate sections here. Each section should be fairly independent.

5.1 Archives

ArX uses archives to store all of the revisions of a project. As explained earlier, archives have a symbolic name (like `dcoyote@yoyodyne`), and an address (like `/home/dcoyote/yoyodyne`). You can have multiple archives on the same machine or multiple machines. `make-archive` creates archives and registers them for you. `archives` lists, registers and unregisters archives. If you need to move an archive, you only need to physically move or copy the archive directory to its new location and re-register the archive.

Most of the time, you do not have to register archives with `archive`s. You can browse archives and get revisions by specifying the complete URI, and that will register the archive for you.

5.1.1 Branches and Revisions

Within an archive are different branches. Branches are a basic way of splitting up work so that people are free to work out improvements without directly upsetting the main development branch. Branches have a hierarchical structure, and can be any number of levels deep. For example, gcc developers might set up a branch called `gcc`. Someone else might be working on a new parser, so they make a branch called `gcc.new-parser`. During the course of their work, the developers working on the new parser might make a branch for improving compilation speed called `gcc.new-parser.speed`. They might make another branch for handling java and call it `gcc.new-parser.java`. The branch names are purely for human consumption, and do not enforce any real relation between branches. For example, `gcc.new-parser.java` might be completely unrelated to gcc or the new parser at all.

Within each branch are revisions. These are numbered starting from zero. A revision is a snapshot of the state of a project. For example, revision 66 might be the project just after some speed improvements have been implemented. Revision 75 might be the project once

all of the bugs in the speed improvements are worked out. Revision 76 might be the project once the docs are updated to reflect the new speedups. And so on. A revision is specified with a leading comma "," to distinguish it from a branch. So revision 66 of `gcc.new-parser` would be `gcc.new-parser,66`.

To summarize, the complete syntax for specifying a project is

```
archive/branch.subbranch,revision
```

If you have defined your default archive, you can omit the archive. There are a number of cases where you may want to specify just an archive. If it is possible for the archive to be confused with a branch or revision, you must follow the archive name with a slash "/". For example, to browse the contents of archive `spanza@example.org`

```
arx browse spanza@example.org/
```

Otherwise, ArX will think that you are trying to browse the `spanza@example.org` branch in your default archive.

5.1.2 Cached Revisions

Arx does not store the full text of all revisions in the archive. Instead, it currently stores the first revision and subsequent patches. This can be quite slow. For example, if you have 1000 revisions, each time you `get` the latest revision, ArX has to get and apply 999 patches to get to the most recent revision. For that reason, you can cache revisions in the archive. Running "`archive-cache --add`" will create a pristine tree of the latest revision and store it in the archive. This has to do all of the patching, but subsequent `get`'s won't have to. This uses up additional space in the archive, because it is storing a tarball of an entire project tree and all of the patches. If you need to reclaim the space, "`archive-cache --delete`" will remove it. Finally, "`archive-cache`" without any options will tell you which revisions have been cached.

5.1.3 Remote Archives

Remote archives are simply archives that are not accessible through the local filesystem. In practice, remote archives are the principal method for distributing software through ArX. For example, remote archives can be pushed to (e.g. mirroring a local archive to a web server) or pulled from (e.g. to download software from that web server). ArX uses the gnome-vfs libraries to access the remote archives over standard networking protocols. That means that if gnome-vfs can see an archive, then ArX can as well. In particular, ArX can access remote archives using http with webDAV, ftp, ssh, and sftp[1]. In addition, if you can not install webDAV, there is an option to use http with explicit lists.

The first thing that you have to do is set up the (s)ftp, ssh, or http server on the remote machine. ArX does NOT have to be installed.

[1]For sftp to work, you must have auto-login enabled.

5.1.3.1 HTTP with webDAV

There are two ways that http access can work. ArX needs to list directories, and plain http does not provide that. HTTP with webDAV is the recommended and most reliable way.

To configure webDAV with apache, this usually involves installing the mod_dav module. This will work with Apache 1.3 or later. It does not require Apache 2. Then you have to add something like the following to the conf file for apache:

```
<Directory /home/*/public_html>
    DAV On
    AllowOverride FileInfo AuthConfig Limit
    Options MultiViews Indexes SymLinksIfOwnerMatch IncludesNoExec
    <Limit GET POST OPTIONS PROPFIND>
        Order allow,deny
        Allow from all
    </Limit>
    <Limit PUT DELETE PATCH PROPPATCH MKCOL COPY MOVE LOCK UNLOCK>
        Order deny,allow
        Deny from all
    </Limit>
</Directory>
```

You might have to change the first line of that to make it point to where your archives are.

5.1.3.2 HTTP with Explicit lists

If you are unable to install webdav support on your server, you can also generate .listing files that contain a listing of a directory. You do this with `update-listing`. For example, once you have created an archive, you can tell ArX to keep the .listing files to the archive up-to-date with a command like

```
arx update-listing -a sftp://dquixote@example.org/public_html/archive
```

For long latency links, this can significantly increase the time to commit and mirror. If you no longer need to keep the .listing files up-to-date, then

```
arx update-listing -d sftp://dquixote@example.org/public_html/archive
```

will stop ArX from updating them.

5.1.3.3 Accessing the Archives

To access archives, just use the ordinary URI notation. Specifically

```
ftp://[user@]host/dir
sftp://[user@]host/dir
ssh://[user@]host/dir
http://[user@]host[:port]/dir
https://[user@]host[:port]/dir
dav://[user@]host[:port]/dir
```
[2]

ArX saves the locations in your `.arx` directory. For ftp and http, passwords are still transferred in plain text. So securely writing to remote archives requires you to use sftp, ssh, or https. Note that the same archive can be registered using different protocols. For example, suppose that you have a website on the machine `example.org` rooted at `/home/dcoyote/public_html/archive`. Through a web browser, it appears at `http://example.org/~dco` Since you can log in to the sftp server, you can register the archive as

```
arx archives --add sftp://dcoyote@example.org//home/dcoyote/public_html/archive
```

while someone who wanted just read access could register it as

```
arx archives --add http://example.org/~dcoyote/archive
```

Note that ArX automatically gets the name of the archive from the archive itself. In fact, in general you do not have to register archives at all, since ArX will automatically register them for you. For example, to browse the previous archive, you can type

```
arx browse http://example.org/~dcoyote/archive/
```

The trailing slash "/" is required so that ArX doesn't look for an archive at

```
http://example.org/~dcoyote/
```

with a branch named `archive`.

5.1.4 Mirrors

Suppose that you do all of your work on a laptop, but you also have access to a web server. To share your work with the world, you want to copy your laptop archive to the web server. Alternately, suppose that someone else has published an archive. You would like to have a local copy on your laptop for when you don't have access to the network. You can use mirrors to manage copies of archives.

Mirrors are not true copies, in that there are certain restrictions when using them. In particular, you can not commit new revisions to an archive. This prevent the case where one person commits a revision to the master archive, and another person commits a revision with the same name to the mirror. Having two different revisions with the same name but different contents will cause confusion, so ArX prevents it.

[2] This is only to access webdav repositories using gnome-vfs 2.10 or greater. Previous versions of gnome-vfs just use the http:// notation.

5.1.4.1 Publishing a local archive

As a concrete example, suppose you have an archive locally and a remote machine that you can access through sftp which also serves as a web server. Then the local archive named `dcoyote@example.org-archive` might be at

```
file:///home/dcoyote/archive/
```

The remote archive could be at

```
sftp://dcoyote@example.org//home/dcoyote/public_html/archive/
```

and it can also be accessed by the web at

```
http://example.org/~dcoyote/archive/
```

To create the remote mirror, it is

```
arx make-archive --mirror dcoyote@example-archive \
  sftp://dcoyote@example.org//home/dcoyote/public_html/archive
```

Since the remote archive will also be available over plain http, you need to tell ArX to update the .listing files

```
arx update-listing -a sftp://dcoyote@example.org//home/dcoyote/public_html/archive
```

Now if you look at the archive registration, you will see

```
$ arx archives dcoyote@example.org-archive
dcoyote@example.org-archive
  file:///home/dcoyote/archive
  sftp://dcoyote@example.org//home/dcoyote/public_html/archive
```

So the single archive dcoyote@example.org–archive has two locations associated with it. The first listed location will always be the one used for `get`, `commit`, etc., unless you specifically use the other uri. For example,

```
arx get dcoyote@example.org-archive/foo
```

will use the local (file:///) archive to complete the get. If you want to test the mirror, you can use the uri instead of the name

```
arx get sftp://dcoyote@example.org//home/dcoyote/public_html/archive/foo
```

and ArX will get the `foo` project from the remote mirror.

Finally, to populate the mirror, you use the `mirror` command and specify the source and destination

```
arx mirror dcoyote@example-archive/ \
  file:///home/dcoyote/archive \
  sftp://dcoyote@example.org//home/dcoyote/public_html/archive
```

This can get quite tedious to type, so you can shorten the uri's as long as they are unique. That is

```
arx mirror dcoyote@example-archive/ file sftp
```

will do the same thing, as will even

```
arx mirror dcoyote@example-archive/ f s
```

By default, ArX will mirror everything from one archive to the other. You can restrict what will be mirrored by adding it to the archive name. So if you have the projects `foo` and `bar`,

```
arx mirror dcoyote@example-archive/foo file sftp
```

will only mirror project `foo`.

Finally, for other people to access it, they will register the archive with the http uri

```
arx archives -a http://example.org/~dcoyote/archive
```

You can also register this location, in which case the output of `arx archives` would be

```
dcoyote@example--projects
  file:///home/dcoyote/archive
  sftp://dcoyote@example.org//home/dcoyote/public_html/archive
  http://example.org/~dcoyote/archive
```

This might be useful to check that the .listing files have been updated correctly.

5.1.4.2 Making a local copy of a remote archive

Another case where mirroring might come in handy is if there is a remote mirror that you want to make a local copy of. Working from the above example, suppose you are Sancho Panza, and you want to have a local copy of Don Quixote's archive. You might do this so that you don't have to wait on the network, or you might want to work where you are disconnected from the network entirely. So you would start with an archive registration like

```
dcoyote@example-archive
  http://example.org/~dcoyote/archive
```

To make a local archive, you might do something like

```
arx make-archive --mirror dcoyote@example.org-archive \
  file:///home/spanza/dquixote-archive
```

Then populating it would be

```
arx mirror dcoyote@example.org-archive/ http file
```

This will give you an archive registration like

```
dcoyote@example-archive
  http://example.org/~dcoyote/archive
  file:///home/spanza/dquixote-archive
```

But this is not quite what you want. When ArX does a `get`, it will default to looking at the remote archive. To make ArX default to looking at the local mirror, you can use the `--make-default` option to `archives`

```
arx archives --make-default dcoyote@example--projects file
```

Once again, you can abbreviate the uri. This will put the `file:///` uri first, so the registration will become

```
dcoyote@example--projects
  file:///home/spanza/dquixote-archive
  http://example.org/~dcoyote/archive
```

5.2 Branching and Merging

Suppose someone (Bob) is writing a sorting program, such as the unix sort(1). Bob just started, so he has only implemented a generic bubble sort. That works well enough for Bob, so he is now concentrating on improving the option handling. Alice, on the other hand, likes Bob's program, but needs a faster sorting algorithm. So she wants to work on improving the sorting algorithm while Bob works on the option handling. Eventually, either Bob will merge Alice's work back into the original program, or Alice will merge Bob's work into her version. ArX handles this sort of situation with branches and merges.

5.2.1 Initial Branching

So let's start again from the beginning. Bob has written a sorting program and published it as `bob@foo.org/sort.bob`. It currently has 23 revisions from `bob@foo.org/sort.bob,0` to `bob@foo.org/sort.bob,22`., as shown in Figure 5.1.

Alice wants to start from the most recent revision, 22, to implement the new sorting algorithm. To do that, she creates a branch in her own archive alice@bar.org. She will create a branch called sort.bob.quick to denote that she is working on a quicksort implementation. To create the branch, she starts by typing

```
arx get bob@foo.org/sort.bob sort_quick
cd sort_quick
arx fork alice@bar.org/sort.bob.quick
```

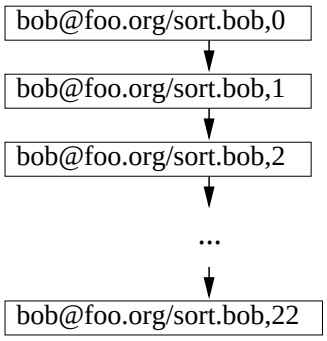

Figure 5.1: Bob's original revisions

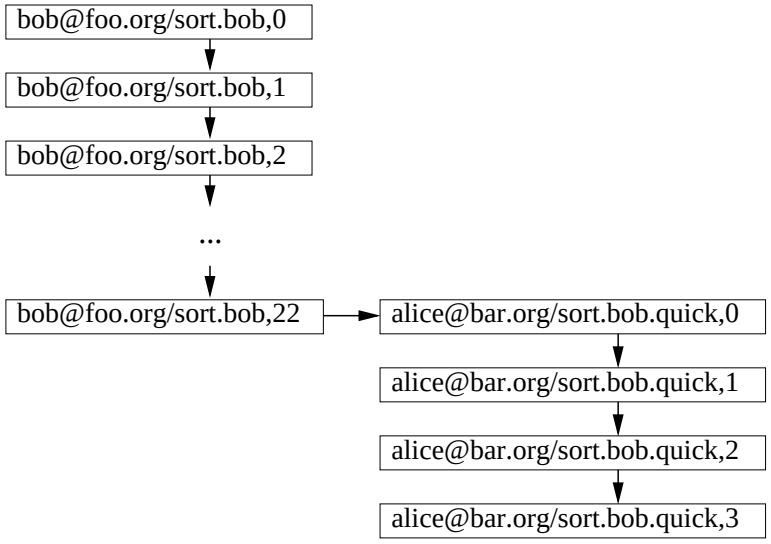

Figure 5.2: Alice's branch

This will create a directory `sort_quick` with the new branch, but not commit anything to the archive. To actually create the new branch in the archive, she just commits

```
arx commit -s "branch to implement quicksort"
```

Now she can use the usual commands to make successive revisions, giving Figure 5.2.

5.2.2 Merge

So Alice is happily hacking along, ripping out the bubble sort and implementing the new quick sort. In the meantime, Bob has not been idle. He implemented some fancy new option parsing with five new revisions, 23, 24, 25, 26 and 27. Alice is not entirely sure what Bob has been up to, but she can find out with `missing`. She types

```
arx missing bob@foo.org/sort.bob
```

and ArX will print out all of the patches that she doesn't have yet (patches 23, 24, 25, 26, and 27). She looks at those changes, and decides to incorporate all of those changes into her branch. So she merges those changes in

```
arx merge bob@foo.org/sort.bob
```

There are a number of ways that changes can conflict (see Appendix B). By default, ArX uses a three-way merge to apply the changes in Bob's tree to Alice's tree. ArX can also merge by looking at the patches that Bob has applied to make `bob@foo.org/sort.bob,22` become `bob@foo.org/sort.bob,27`, gathering it into one big patch and applying it to Alice's `sort_quick` tree.

```
arx merge --algo patch bob@foo.org/sort.bob
```

Either method is fairly sophisticated, handling a number of cases automatically. For everything but modification of file contents (i.e. renames, deletes, metadata changes), the two methods are identical. For example, even if a file has moved, ArX will still know which file to modify. But sometimes there are problems. Bob may have made incompatible changes to a file that Alice modified.

The three-way merge is slightly better at avoiding conflicts, so it is the default. But when conflicts are inevitable, the real difference between the two methods is how conflicts are resolved. When there are conflicts in file contents, the three-way merge method will leave four files in the project tree: The original tree file, the sibling file, the ancestor file, and the output of diff3 when it tried to merge the three. The diff3 file has inline conflict markers similar to what CVS conflicts give. To resolve the conflict, you can edit the diff3 output and remove the other three files. You can also run a GUI three-way merge tool on the three files: tree, ancestor, and sibling. You can even have ArX pop up a merge tool automatically when it detects conflicts. See Appendix C for details.

When there are conflicts using the big cumulative patch method, ArX leaves three files: the original tree file, a modified file with as many of the hunks of the patch applied as possible, and a file with the rejected hunks.

For any conflict, ArX will print out an error message detailing what went wrong. Alice can retrieve these messages with `arx resolve`. Once Alice cleans up all of the problems, she must tell ArX that the conflicts have been resolved with `arx resolve`. Only then will she be able to commit

```
arx commit -s "Merge from sort.bob,27"
```

Her branch will then incorporate Bob's improvements, giving Figure 5.3.

5.2.3 Replay

The `merge` command looks at the differences as a large, amalgamated whole. Sometimes, it can be advantageous to consider differences patch by patch. This is what replay does. If Alice had instead typed

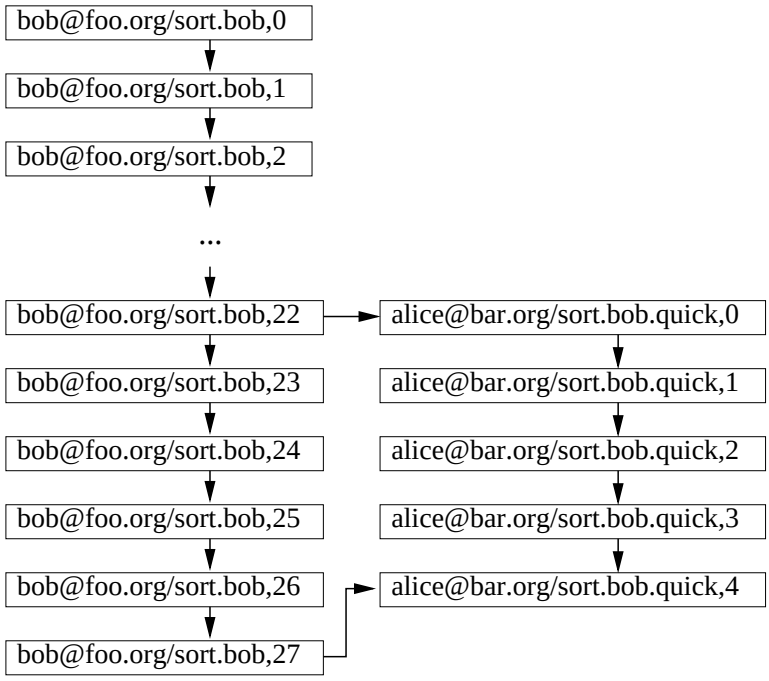

Figure 5.3: Alice's branch updated with Bob's improvements

```
arx replay bob@foo.org/sort.bob
```

then ArX would instead have tried to first apply patch 23 onto Alice's tree, then patch 24, then patch 25, etc. If, along the way, any of those patches caused a conflict, ArX stops with that patch and lets you fix up the tree before continuing. For example, suppose patch 24 had a conflict. `sort_quick` will contain the result of ArX's attempt to patch up to patch 24. Once Alice fixed up the `sort_quick` directory, she can just repeat the same command

```
arx replay bob@foo.org/sort.bob
```

and ArX will attempt to continue the update. If there are other conflicts, Alice can continue fixing conflicts and repeating the update until she reaches Bob's current version. Assuming that Alice resolved conflicts the same way, this should give exactly the same result as Figure 5.3.

However, replay also offers the possibility of selectively applying patches. Suppose Alice didn't like all of Bob's patches, but only liked patches 25 and 27. She could have incorporated those changes, and only those changes, with the commands

```
arx replay --exact bob@foo.org/sort.bob,25
arx replay --exact bob@foo.org/sort.bob,27
```

This will apply patches 25 and 27 to the tree. She can also do it in one go by putting the patches names in a file and using the `--list` option.

Unfortunately, once she uses selective patching, she can't use `merge` or `replay` in their generic form anymore. They will both want to incorporate the patches that she deliberately

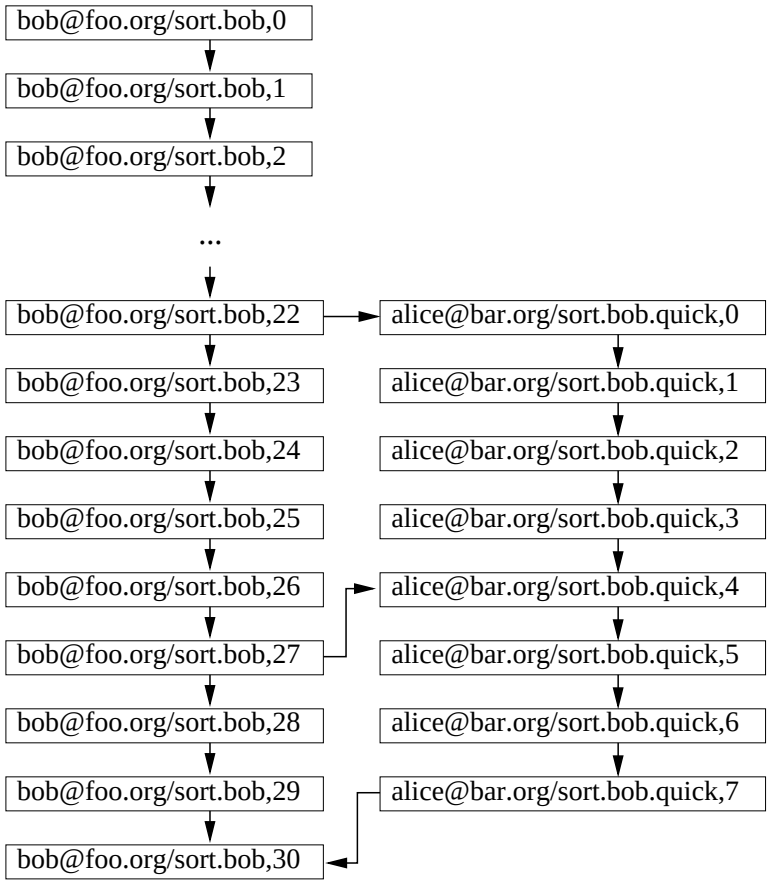

Figure 5.4: Bob's branch star-merged with Alice's

skipped. ArX currently does not have a means of marking certain patches as unwanted. If she wants to continue to get updates from Bob, she will always have to use the `--exact` or `--list` options.

5.2.4 Merging Back

Alice can continue hacking, using `merge` or `replay` to update her branch with Bob's changes. Eventually, Bob may want to integrate Alice's changes. This is as simple as

```
arx merge alice@bar.org/sort.bob.quick
```

This will incorporate all of Alice's patches. ArX is smart enough to know that it really only needs the patches after patch 27 from Bob, because `alice@bar.org/sort.bob.quick,4` integrated Bob's branch up to patch 27 into Alice's. As usual, this patching may cause a conflict. Once that is cleaned up, a commit will lead to Figure 5.4.

As Alice and Bob continue improving their respective branches, they can continue to merge with each other with `merge`.

5.2.5 Bug Fix Branches

There are other reasons for making a branch. If you have made a release of your software, and want to fix issues in that release without affecting current development, then you can branch from that release. For example, suppose that Bob had made a release of `bob@foo.org/sort.bob,20`[3]. Later, after Bob has merged in the support for Alice's work on quick sort, someone finds a bug in his implementation of bubble sort (how embarrasing!). Bob can't make the fix in the current line of development, because the bubble sort has been removed. Bob also can't release the current line of development because the project is not in a releaseable state. Instead, he branches from the release and fixes the bug on the branch. Specifically

```
arx get bob@foo.org/sort.bob,20 fixed
cd fixed
(fix the bubble sort bug)
arx fork bob@foo.org/sort.bob.fixed
arx commit -s "Fixed the bubble sort"
```

Bob can now make a new release with `bob@foo.org/sort.bob.fixed,0`, and it will only have the fix to the bubble sort bug.

If it turns out that the bug has already been fixed in the current line of development, Bob can pull in just that change. For example, if there is a typo in the help screen that has been fixed in `bob@foo.org/sort.bob,30`, then Bob can apply just that patch with

```
arx replay --exact bob@foo.org/sort.bob,30
```

See also Section 5.8.2 for how to apply patches with even finer granularity.

5.3 Remote Cooperation and Publishing Your Work

5.3.1 Tags

5.3.1.1 Release Markers

If you wish to distribute your work, you can create and update a mirror as described in section 5.1.4. Then other people can register your archive and get the latest revision using `arx get`. However, people may not want to continuously follow every little spelling and off-by-one bugfix. They only want significant, well-tested improvements. You can accommodate them by using symbolic names to mark certain revisions as stable. For example, consider the situation given in Figure 5.1. If Bob is happy with the state of the tree at that point, then he can create a release marker with

```
arx tag bob@foo.org/sort.bob.release bob@foo.org/sort.bob
```

[3] It is usually best to use tags (Section 5.3.1) to mark releases rather than using a specific revision number.

This creates a revision, `bob@foo.org/sort.bob.release,0`, which acts as a symbolic name for the last revision of `bob@foo.org/sort.bob` (in this case, that would be `bob@foo.org/sort.bob,22`). Bob can run this command whenever he wants to make a release. An interested user can get the latest release with

```
arx get bob@foo.org/sort.bob.release
```

Because `bob@foo.org/sort.bob.release` is just a symbolic name, you can not fork directly from it. Rather, you must fork from the referenced revision.

You can also use tags to just give a different name for a particular revision. For example, suppose you had a branch `myproduct`, and your marketing experts wanted to name the next version `v0.0.1`. Then

```
arx tag myproduct myproduct.v0.0.1
```

will create a revision that you can get with

```
arx get myproduct.v0.0.1
```

5.3.1.2 Collections

You can also use `tag` to mark a collection of projects. So if you have the directory structure

```
foo ----> Contains the project foo.main
foo/bar ----> Contains the project bar.main
```

then you can mark the whole collection of projects with

```
arx tag foo.collection foo.main bar.main bar
```

That is, you first specify the head project (`foo.main`). For the tail projects, you specify the project name (`bar.main`), and the subdirectory that it goes into (`bar`). Then

```
arx get foo.collection
```

will download both `foo.main` and `bar.main`, and put `bar.main` into the `bar` subdirectory of `foo.main`.

You can have as many sub-projects as you wish. For large, complicated projects, you can read in a list of projects from a file. The format is

```
tag-name
head-project
sub-project sub-dir
sub-project sub-dir
...
```

Then you read it in with

```
arx tag -f FILE
```

5.3.1.3 Floating Tags

With these large, complicated projects, another problem emerges. When you define a tag, it will point to a particular revision. In the previous section, running

```
arx tag foo.collection foo.main bar.main bar
```

will create a tag `foo.collection,0` that points to the last revision of `foo.main` (perhaps `foo.main,12`) and the last revision of `bar.main` (perhaps `bar.main,7`). If you continue to work on `foo.main` and `bar.main`, then running

```
arx get foo.collection
```

will always give you `foo.main,12` and `bar.main,7`. On the other hand, if you use a floating tag

```
arx tag --float foo.head foo.main bar.main bar
```

then `foo.head` will always point to the latest revisions of foo.main and bar.main. You can now always get the latest version of the entire collection with

```
arx get foo.head
```

If you just want to update a copy of the tree, then it is just

```
arx merge foo.head
```

5.3.1.4 Limitations

Tags will show up as ordinary branches in `arx browse`, but they have a few restrictions. In particular, `tree-cache`, `replay`, `fork`, `file-diff`, `file-orig`, `file-undo`, and `get-patch` will not work with tags. `diff` does not work directly with tags, but there is a `--recursive` option to handle collective tags. `archive-cache`, `get`, `export`, `missing`, and `merge` will work even with collective tags. So `get` gets all of the different projects, `merge` updates the main project and subprojects, etc. `merge` also has a `--recursive` option, which updates all of the subdirectories, not just the ones listed in the tag. The `--recursive` is equivalent to running `merge` in each subdirectory. So it will not update a subdirectory to a new branch, while using merging with a tag could.

5.3.2 export

You may decide that you do not want to make people use ArX just to get a revision. Even without other people involved, you may want to use your work in different environments that do not have ArX installed. `export` can create a tree without any of the ArX control files and, for your convenience, a tarball of this tree. For example, if the latest revision of hello.main is patch 6, then

```
arx export --tarball hello.main hello
```

will create a tarball in the current directory named hello.6.tar.gz. If you desire, you can include GNU-style changelogs with the `--changelog` option. `export` will also work with tags5.3.1.

5.3.3 Applying patches directly

Sometimes it is useful to generate and apply patches directly. For example, you might fix a bug in a project, but have no means of publishing the archive. So you want to just mail the patch directly to the upstream author.

```
arx diff -o diffdir
```

`will` create a patch in the `diffdir` directory. You can simply tar up that directory and mail it.

If you have made your own archive with several patches, you can still bundle all of the changes together with `diff`. You just supply `diff` with the last upstream revision. For example, given the situation in Figure 5.2, Alice can create a patch of all of the work she has done with

```
arx diff -o diffdir --revision bob@foo.org/sort.bob,22
```

Alternately, you may wish to give the patches back piecemeal, so that the upstream author can take only what they want. You can get a specific patch with `get-patch`. So again with the example in Figure 5.2, Alice can get all of the indiviual patches with

```
arx get-patch alice@bar.org/sort.bob.quick,0 alice.0
arx get-patch alice@bar.org/sort.bob.quick,1 alice.1
arx get-patch alice@bar.org/sort.bob.quick,2 alice.2
arx get-patch alice@bar.org/sort.bob.quick,3 alice.3
```

This will put the patches into the directories `alice.0`, `alice.1`, `alice.2`, and `alice.3`.

If you want to put together some, but not all, of the patches, you can create a new project tree with the upstream author's latest version. Then you can apply the specific patches with `replay` and `diff` will produce a patch that encompasses all of the changes. With the Alice/Bob example

```
arx get bob@foo.org/sort.bob,22 sort
cd sort
arx replay --exact alice@bar.org/sort.bob.quick,1
arx replay --exact alice@bar.org/sort.bob.quick,3
arx diff -o patch_1_3
```

This will put a patch in the directory `patch_1_3` which agglomerates the patches `alice@bar.org/sort.bob` and `alice@bar.org/sort.bob.quick,1`.

On the receiving end, if the original author has a project tree in directory `foo` and the patch unpacked into the directory `diffdir`, then

```
arx dopatch diffdir foo
```

will apply the patch.

5.3.4 Multiple commiters (a la CVS)

If you have a particularly large, active project, you may have many different people updating various parts concurrently. You want to allow the main development branch to be updated by multiple people. This is the usual style of development with large projects using CVS. There are a few ways to do this in ArX.

The first is to just make the archive directly writeable by all of the developers. If you are all in the same place, you might do this with NFS. However, you do have to be careful with permissions and umask. Otherwise, one developer committing changes may make it impossible for other developers to commit.

If you are in separate places, you can give all of the developers accounts to a shared filesystem, once again being careful about permissions and umask. If permissions and umask problems are insurmountable, or you do not wish to make new accounts for every person who is vaguely interested in the project, then you can make a single account that all of the developers can use to update the archive with sftp. That means making a new account for each project. It also makes it more difficult to audit the activity in the source tree, since there is only one user-id associated with all of the changes.

One solution is to designate a person as an integrator. Developers branch off of the integrator's main line of development. When the developer is ready, they send a merge request to the integrator. The integrator applies the change to a test tree, runs any tests, and commits. This solution is nice in that it does not require giving out accounts to anyone. However, with a busy project, the integrator can get overwhelmed.

Fortunately, this process can be automated with a patch queue manager (PQM). A nice PQM is bundled with ArX in the `tools/pqm` directory. The idea is to have a special PQM account that manages the main line of development. Then developers branch off of this main line of development. When the developer is ready, they send a merge request to the PQM account (e.g. through a signed email). The PQM account attempts the merge, and, if successful, commits the change. If the merge fails because of conflicts, then nothing is committed. More documentation can be found in `tools/pqm`.

5.4 Reverting development

Sometimes, you want to undo some of the changes that you have made.

5.4.1 Before you commit

If you haven't committed the change to the archive and you only want to revert one file, you can use `file-undo`. That is, to undo the changes made to file `foo`, you would type

```
arx file-undo foo
```

However, `file-undo` will not work properly if you deleted the file with "`arx rm`" or moved it with "`arx mv`". You can also decide to undo a file back to a particular revision by specifying that revision

```
arx file-undo foo hello.main.1.0,11
```

Using a revision other than the most recent may require ArX to get that old revision, which can be time consuming. If this is a problem, you can add that revision to your pristine trees.

If you want to undo the changes made to a number of files, or you used "`arx rm`" or "`arx mv`" on a file you want to undo, then you have to use `undo` with those paths as extra arguments. That is, if you want to undo the files `foo` and `bar` plus everything in the directory `bat`, you can use the command

```
arx undo foo bar bat/
```

If the changes in the directory `bat/` depend on changes elsewhere, then ArX will let you know what you need to include. If you want to undo everything that has been done since the last commit, just use `undo` without any arguments.

`file-undo` will store a copy of the old file in „file-name, so you can get back your changes by copying that file back. `undo` will store the changes in a directory „undo-N, with N being the smallest number not already taken. To get the changes back, you can use `redo`. If you don't give `redo` any arguments, `redo` will just use the largest numbered „undo-N directory. Of course, if you decide that you really don't need the changes, you can simply delete the „file-name files and „undo-N directories. ArX will never delete them itself.

In summary, `file-undo` is nice for changes to a single file, because ArX keeps a complete copy of the modified file around. `undo` only keeps a copy of the differences between the original file and the modified file. `file-undo` also works for any revision, while `undo` only works for the current revision. However, there are times when `file-undo` will not work, while `undo` always works.

5.4.2 After you commit

If you have already committed the changes to an archive, then there are two ways of reverting those changes: Non-destructive and Destructive.

5.4.2.1 Non-destructive revert

If you decide that you need to revert changes that have already been committed to an archive, then you need to use the `--add` option to `history`. This will make it look like a tree has the patches for a particular revision without actually applying them. So, for example, suppose you really want the most recent revision to look like hello.main,12, but you made some ill-considered changes in patches 13 and 14, then you could run

```
arx get hello.main,12 hello
cd hello
arx history --add hello.main
arx commit -s "Reverted patches 13 and 14"
```

That will create revision patch 15 which will look exactly like revision patch 12 except for the history. Then, if it turns out later that patches 13 and 14 were not such a bad idea, you can still get them.

This also lends itself to more complicated scenarios, where not everything in patches 13 and 14 was bad. For example, suppose the changes to directory bat/ were good, but everything else was bad. Then you can use the sequence

```
arx get hello.main,12 hello
cd hello
arx history --add hello.main
arx undo bat/
arx commit -s "Removed everything in patch 13 and 14 except changes to bat"
```

This preserves the changes to `bat/` but not anything else.

Finally, you can use `history --add` just to synchronize development. Consider a branch hello.branch that, for whatever reason, you want to make exactly the same as hello.main. Perhaps all of the changes that were in `hello.branch` got integrated in various ways into `hello.main`, and you now want to re-sync `hello.branch`. You can accomplish this with

```
arx get hello.main hello
cd hello
arx history --add hello.branch
arx tree-version hello.branch
arx commit -s "Synchronize with main"
```

Most of the time, though, you would probably just abandon the old branch and make a new one. Otherwise, people may get confused by a branch that changes meaning.

5.4.2.2 Destructive revert

You may have accidently committed a file that is extremely large, has corporate secrets, is illegal to distribute, etc. In those cases, you will want to do a destructive revert of your changes and reclaim the space. ArX does not allow you to just remove a revision. That is because someone may have forked from there, and if you replace one revision with a different one, then ArX will get very confused. Essentially, ArX is trying to keep you from changing history.

As an example, suppose that revision foo,23 has one of these undesirable files. You can issue the command

```
arx delete-revision foo,23
```

and ArX will print out a formatted version of the log and prompt you to continue. If you do so, that revision will be replaced with an empty revision. It will have a log just stating that it has been reverted. To continue development, `fork` from the previous revision

```
arx get foo,22 new_foo
cd new_foo
arx fork foo
```

and work in the `new_foo` directory. If you didn't notice the problem until after revisions were added past `foo,23`, you can replay those patches

```
arx replay --exact --dir new_foo foo,24
arx replay --exact --dir new_foo foo,25
arx replay --exact --dir new_foo foo,25
...
```

If you had just used merge, you will get the deleted log message. Not the end of the world, but a minor annoyance.

If you want to delete an entire branch, then `delete-branch` will remove everything, leaving no traces of that branch. For example,

```
arx delete-branch foo
```

will delete everything in the branch `foo`, including any sub-branches such as `foo.bar`, `foo.bar.baz`, etc. This is a powerful command, and ArX prompts you before deleting.

5.5 Properties

5.5.1 Preserving File Permissions

ArX allows you to assign arbitrary properties to paths. The primary application for this within ArX is to version permissions. For example, if you want to make sure that the file `foo` will have its executable bit set, then the command

```
arx property --set arx:user-exec true foo
```

will ensure that. At present, the following properties have predefined meanings

```
arx:user-read
arx:user-write
arx:user-exec
arx:group-read
arx:group-write
arx:group-exec
arx:other-read
arx:other-write
arx:other-exec
```

If any of these properties are set to true or false, then when that path is checked out (e.g. with get), the appropriate permission bit is set (assuming the file system can accomodate it).

5.5.2 User Defined Properties

You can also define your own properties. For example, you can assign a property that tells you what kind of license a file is covered by. If you type

```
arx property --set license GPL foo
arx property --set license BSD bar
```

then the license for file `foo` is set to the GPL and the license for `bar` is set to the BSD license. Note that, while the properties can be arbitrary, they are designed to work well when they are small.

5.5.3 End-of-Line Conversion

By default, ArX does not do any conversion of the end-of-line markers used in files. In the future, ArX may use the arx:eol-style property to do something similar to what Subversion does[4].

5.6 Hooks

One approach to quality control is to have a modified project tree go through a series of automated tests before the modifications are stored in the archive. A simple example is to make sure that the modified tree will build. Once the patch has gone through, you may wish to automatically perform various actions, such as sending mail about a patch to interested parties. ArX itself uses this feature to update the arx-changes list.

ArX supports these two needs through hooks. To use hooks, you create an executable file in `~/.arx/hooks`. It can be a shell, Python, or Perl script, or even a full blown C, C++, Java or Lisp application. ArX invokes the hook both just before and just after it has altered an archive by adding categories, branches, versions, or revisions. This occurs when you invoke `commit`, `tag`, or `mirror`. ArX calls the hook with two arguments. The first argument is either `pre` or `post`, indicating that the hook is being called either before or after altering the archive. The second argument is one of `make-branch` or `make-revision`, indicating what ArX is about to do or has done. That is, the call syntax looks like

```
~/.arx/hooks (pre|post) make-(branch|revision)
```

In addition, ArX sets the environment variables `ARX_TREEROOT` to the root of the project tree (if applicable), `ARX_PREVIOUS_ARCHIVE`, `ARX_PREVIOUS_ARCHIVE_URI`, `ARX_PREVIOUS_BRANCH`, and `ARX_PREVIOUS_REVISION` to the archive, archive uri, branch, and revision of the previous revision, and `ARX_ARCHIVE`, `ARX_ARCHIVE_URI`, `ARX_BRANCH`, `ARX_REVISION`, and to the archive, archive uri, branch, and revision involved. These can be queried to customize how the hook behaves. As an example, the following shell script will send email about new categories, branches, versions, and revisions in the `wlandry@ucsd.edu--arx` archive to the arx-changes list

[4] http://svnbook.red-bean.com/en/1.1/svn-book.html#svn-ch-7-sect-2.3.5

```
#!/bin/sh
# Simple mail of patch log
pre_post=$1
action=$2
if test $pre_post = "post" ; then
  if test $ARX_ARCHIVE = "wlandry@ucsd.edu--arx" ; then
    if test $ARX_ARCHIVE_URI = \
        "sftp://landry@superbeast.ucsd.edu//home/landry/public_html/ArX/wlandry" ;
      if test $action = "make-branch" ; then
        printf "$ARX_ARCHIVE" | mail -s \
               "New Branch: $ARX_BRANCH" arx-changes@nongnu.org
      fi
      if test $action = "make-revision" ; then
        arx log --remote --formatted --branch $ARX_REVISION | mail -s \
               "New Revision: $ARX_REVISION" arx-changes@nongnu.org
      fi
    fi
  fi
fi
```

The hook script is executed within the current directory. This script will be executed whenever you alter any archive, so a long complicated script will slow these actions down. When invoked before altering the archive, ArX waits for the hook script to return and aborts if it returns non-zero. When invoked after altering the archive, ArX executes the hook script in the background and ignores the return code. Post-commit hooks are never guaranteed to be invoked. A well timed interrupt could let the transaction finish but prevent the hook from running.

5.7 Patch Logs and Changelogs

When committing a change, ArX needs a log file with a Summary: field. If you use the `-s` option to commit, then ArX will create a log file for you that contains that field. However, you can also create your own log files with custom headers. There are some reserved headers (such as Standard-date:, Renamed-files:, etc.) listed in the help for `log`, but otherwise you can define any header you like. The log file uses an RFC-822 style format. A colon separates the header and the field, and the field is terminated by a newline that is not followed by a tab. The body is separated from the headers by a blank line. As an example,

```
Summary: Frozzled the foo
Mail-results-to: don@example.org, sancho@example.com,
    dulcinea@example.net
Bug-Number: 1605

The foo was blarged by the bar, so I had to frozzle the foo in order to
unmome the borogoves.
```

You can then specify that log file with the `--log-file` option to commit.

ArX adds some reserved fields and stores the log as part of the patch. These logs then become part of the revision. When you make a branch, your logs for that new revision appear in the project tree. You can see what versions have gone into a project tree with `history`. For each of the versions that it lists, you can find out which patches are included with `log`. `log` also lets you look at specific headers. A simple example is to look at the `New-files:` field for all of the patches for the current version of the tree

```
arx log --header Revision --header New-files
```

The `Revision` header is included because otherwise there is no way to tell which new files belong to which revision.

You can also do more complicated things, such as finding when foo.bar.1.0,112 was created and by whom

```
arx log --header Standard-date --header Creator \
  --branch foo.bar.1.0,112
```

You can use the `--remote` option to look at logs for revisions that you don't have in a project tree. For example, if you were unsure whether you wanted to get those revisions at all.

5.8 Making Patches Bigger or Smaller

ArX currently does not support directly breaking up one patch into smaller patches or composing multiple patches into one big patch. You can achieve the same effect through some workarounds.

5.8.1 Selective commits

Suppose you are happily working on one feature, but along the way you notice and fix a bug in unrelated functionality. You would like to separate the bug fix from the ongoing feature work. Usually, the best way to do this is with extra path arguments to `commit`. For example, if the features are in file `foo`, and the bug fix is in file `bar`, then "`arx commit bar`" will only commit the changes in file `bar`. You can also select files that have been added, moved, and deleted. ArX performs thorough checks to make sure that you always commit a valid patch. For example, if you were not careful, you might commit a file that is in a directory that does not yet exist in the archive. If you try to make ArX do this, then ArX will tell you what paths need to be added the argument list.

If the separation between bug fix and feature is not so clean, such as if the changes occur in the same file, then you can use `undo`. You run `undo` on the whole directory, make and commit the bug fix, then `redo` to get back the work you've done. More information on `undo`/`redo` is in Section 5.4.

5.8.2 Breaking up patches

Suppose that someone has created a humongous, all-singing, all-dancing patch that adds 12 features, fixes 30 bugs , and, of course, introduces its own. You are only interested in a particular feature which is localized to files foo1, foo2, foo3, etc. To get just the changes to these files, you can do something like

```
arx replay --exact bar.big-patches,13
arx undo -o foo_undo foo1 foo2 foo3 ...
arx undo
arx redo foo_undo
```

This gets the patch, applies it to your own tree, selectively reverts the feature you want, reverts everything else, and then reapplies the desired feature.

5.8.3 Agglomerating patches

Suppose you have a project `foo.main`, and you want to make a patch that includes patches 122, 133, and 156 all as one big patch. You can do it with something like

```
arx get foo.main,121 foo
cd foo
arx replay --exact foo.main,122
arx replay --exact foo.main,133
arx replay --exact foo.main,156
arx diff -o big_patch --revision foo.main,121
```

This gets revision 121, applies the various patches selectively, and then puts the agglomerated patch into the directory `big_patch`.

5.9 Working with Large Trees

5.9.1 arx edit

By default, ArX is set up to be very careful when looking for changes. This means that ArX has to look at the contents of every file before it can decide whether it has changed. This can be prohibitively slow for large projects. So ArX offers another mode of operation where you can promise not to edit a file unless you specifically tell ArX. This approach is similar to what Perforce and Bitkeeper do.

You can take advantage of this mode of operation by using the `--no-edit` option to `get`. ArX will download the revision and then change the permissions on all of the files to read-only. To edit a file, you have to run `arx edit` and ArX will change the file to writeable. Then, when you run `arx commit`, ArX will once again mark the files as read-only. You do not have to run `arx edit` in order for `arx rm`, `arx mv`, and `arx property` to work.

The advantage of this is that when ArX figures out what you have changed for `diff` or `commit`, ArX only has to look at the short list of files that you have marked (via `edit`, `rm`,

`mv`, and `property`). This can reduce the time for these common operations from minutes to near-instantaneous. However, some people find this mode of operation incredibly annoying. Others hardly notice it. You only need to use it if you are running into problems. In general, if your project tree is in memory, ArX humms right along. However, if the project tree is not in memory, ArX has to load it from disk which can take a rather long time. Whether your tree is in memory depends on your individual work patterns.

If you decide that you want to always work in this mode of operation, you can set to `true` the `no-edit` parameter in `arx param`. Then `get` will always run as if the `--no-edit` option is present.

5.9.2 link-tree

In addition to the `--no-edit` option, there is a `--link-tree` option. It is only useful with the `--no-edit` option. The `--link-tree` option will use hard links when getting a tree, reducing both the space and time required. However, because it links with cached revisions, write permissions in the cache will get modified as well. This means that versioned write permissions will, in general, be unreliable. If it turns out that write permissions are not important for your project (as is often the case), then `--link-tree` could well be a useful option. Like `no-edit`, you can set the `link-tree` parameter in `arx param` to make hard linked trees the default.

5.9.3 Timestamps

Another possible method that could have been used is to save timestamps of files on the initial `get`. Then figuring out whether a file has changed means ArX would only have to look at the timestamp of a file. You can also compare more than just timestamps (e.g. size). This method is very popular, being used by TLA, Darcs, Subversion, CVS, and Stellation. It is a little nicer interface, since you do not have to explicitly mark a file as editable before editing.

However, it falls down on many common filesystems. Many filesystems have a timestamp resolution of one second. That means that if you get a project tree and edit a file all within one second, then that file will not show up as changed. Normally, people can not type that fast, so it is not a problem. However, if you are using some kind of automatic patch robot (as in section 5.3.4), then the robot will create a project tree and apply the patch. Some changes to files may then be committed, and others not. In general, any kind of scripted use can cause these problems.

These problems are not academic. All of the aforementioned version control systems have had problems arising from these inexact timestamps. Darcs even has an `--ignore-times` option, which is great if you remember to use it. Because of this inherent unreliability, ArX does not implement this method.

5.10 Cryptographic Checksums and Signatures

5.10.1 Theory

Once data is stored in an archive, it may become modified or corrupted. These modifications could be accidental (e.g. disk corruption) or intentional (e.g. someone trying to insert malicious code).

To detect these modifications, the first thing that ArX uses is checksums. There are two kinds of entities that get checksums in ArX: patches and revisions. A revision is just a complete source tree, and a patch is what gets you from one revision to another. Patches are simply tar'd, gzip'd file trees, and gzip has its own checksum. Revision checksums are more complicated.

Whenever ArX stores a revision in the archive, it creates a manifest file. The manifest file lists each path in the revision, its properties (set with `arx property`), and a cryptographic checksum[5] of the path's contents. ArX then computes a cryptographic checksum of the entire manifest, and stores that into the archive. When someone downloads a particular revision, ArX recreates the manifest file based on what it has downloaded. ArX then checks the checksum of the newly created manifest file against the checksum in the archive. All of this is completely automatic, and you won't notice it unless something goes wrong.

However, while this may work great for catching errors due to corrupted hard drives and bad memory, it won't stop someone from deliberately inserting malicious code into the archive. They can always replace the checksum while replacing the original patch. To solve this, ArX uses cryptographic signatures.

Once again, both revisions and patches can be signed. Patches are signed directly by storing a detached signature of the patch file in the archive. Revisions are signed indirectly by storing a detached signature of the revision checksum in the archive. In addition, ArX stores in the archive a list of cryptographic keys that are allowed to sign revisions in that archive.

So the first time a person downloads a revision or patch from a particular archive, ArX will download the list of cryptographic keys. ArX will then download the actual revision or patch and check to make sure that it is properly signed by someone in that list.

ArX uses Gnu Privacy Guard (gpg) to do the actual creation and verification of signatures. This has an advantage over other types of signatures (e.g. X.509) in that a number of people already have a gpg key. An X.509 certificate would just be another secret to protect, another password to remember, etc. In addition, your gpg public key may be already be known to the recipient.

For those of you already using gpg, ArX does not use the usual web of trust. If you want to download a revision from a random place on the web, you don't want to have to extend your trust for other things to this particular public key. Moreover, if someone manages to compromise one person's key, they may be able to subvert a larger number of projects. However, this does mean that you should verify the public keys you download.

It should also be noted that, while ArX uses SHA-256, gpg may internally use something weaker (e.g MD5 or SHA-1). If you are concerned, you should consult the gpg documentation

[5] ArX uses SHA-256 for its cryptographic checksum. This checksum has no known weaknesses (as opposed to MD5 or SHA-1), and should be sufficient for the next 50 years or so.

to make sure you are using a secure hash.

5.10.2 Practice

As noted before, you do not need to do anything for ArX to support checksums. ArX will automatically create and validate all checksums and let you know if there are any problems.

To verify signatures of signed archives, you only need to have compiled ArX with gpg support. ArX will automatically download public keys, and download and verify signatures. You can use `arx archives` to see what public keys are associated with an archive and verify that the keys are genuine. You can quickly verify the signatures for all of the revisions in a branch with the sig command

```
arx sig dcoyote@example.orc/hello
```

To sign your own archives is where you have to do some work. Signatures are managed on a per-archive basis. Either everything in the archive is signed, or nothing is. To create an archive that will be signed, use the `--key` option to `make-archive`. For example

```
arx make-archive --key dcoyote@example.org \
         dcoyote@example.org--archive archive
```

The argument to `--key` can also be a gpg fingerprint. If you want every archive you create to be signed, then use `arx param` to set the `gpg-key` parameter to your gpg public key. This will also set what your default key to sign archives will be.

Once you have a signed archive, ArX will ask for your gpg passphrase each time you commit. This means that you will have to type in a passphrase twice each time you commit: once for the patch and once for the revision. That can quickly get tedious. So you can tell ArX to use a program such as quintuple-agent to store your password. For quintuple-agent, that would be

```
arx param gpg agpg
```

Now ArX will use `agpg` when trying to sign and verify revisions. Quintuple-agent also requires you to set up an agent, which you will have to do separately.

If you have already created an archive and you want to make it signed, you first need to add your public key to the archive using a command like

```
arx sig --archive --add dcoyote@example.org--archive/
```

Then you can manually sign each patch and revision with something like

```
arx sig --add dcoyote@example.org--archive/hello,0
```

or just sign all the patches and revisions in a branch with

```
arx sig --add dcoyote@example.org--archive/hello
```

If you have any mirrors, you should delete them and re-mirror.

You can also delete a signature with the `--delete` option. All of these examples will add or remove your default gpg public key set with `arx param`. To add or delete a different key, use the `--key` option.

Finally, you need to let everyone else know that your archive is now signed. Other people accessing the archive will not automatically update the list of keys to trust. So if you try to sign revisions with the new key, they will not validate the signature. They must unregister and reregister the archive.

5.11 Internationalization

ArX takes a laissez-faire attitude to internationalization. In particular, ArX treats everything as a sequence of bytes, and does not attempt to convert anything into a canonical form (e.g. UTF-8). So file contents can be in any encoding, and ArX will not care. If Gnu diff thinks that a file is binary, then ArX will use a binary diff and patch. This prevents automatic merging, but otherwise everything will work fine. Moreover, ArX does not do any line-ending conversions for Windows and Unix clients.

The situation with file and directory names is more complex. ArX uses C Posix API's such as stat() which require null terminated strings. So if your file names have any embedded nulls, you will quickly run into problems. What this means is that if you use UTF-8 everywhere, then you should have no problems. With the various Latin encodings, all of the files will be stored correctly, but they may not display correctly if someone has a different locale.

No guarantees are made for other encodings. In particular, Shift-JIS, Big5, VISCII, and KOI8 will probably have problems. Those encodings use the slash "/" character in a multibyte character, which will make ArX think that the path is referencing a subdirectory.

5.12 Including one project within another

Suppose you have projects foo and bar, and you want to merge project bar into foo. That is, you want all of the files in bar to be present in foo. All you need to do is move all of the files in bar into foo with "`arx mv`". Then you just syncronize the foo tree with bar with `history --add` and then commit. All of the history will be preserved, even if patches are applied from the old project.

5.13 Project Tree Inventories

5.13.1 Inventory Ids

When ArX is looking at a path, ArX wants to assign the path a unique identity that will persist even when the path is renamed. ArX does this with inventory ids. An inventory id is just an alternate name for a path. When a path is first introduced to ArX with "`arx add`", it will have an inventory id associated with it.

Type	Stored in Archive?	Created by	tree-lint warning?
source	Yes	User	No
control	Yes	ArX	No
ignored	No	User	No
unrecognized	No	User/ArX	Yes

Table 5.1: Inventory Types

Inventory ids are contained inside a small file inside the `_arx` directory which you should never deal directly with. When you need to move or delete the path associated with the external inventory id file, you must use "`arx mv`" and "`arx rm`". Otherwise ArX will get confused.

If you do not explicitly add a path, then it will not get archived. `tree-lint` and `inventory` come in handy here. If you do not use ArX to move and delete paths, then ArX will notice when you try to commit and force you to fix it.

5.13.2 Inventory Types

When ArX looks at a project tree, it likes to divide the paths into various types. There are five different types: **nested_tree**, **source**, **control**, **ignored**, and **unrecognized**. A **nested_tree** is merely a project tree within a project tree. The other types require more explanation.

These types come about because ArX has some decisions to make when looking at a file. ArX has to decide whether a file will get stored into an archive. Files classified as **source** or **control** are stored in the archive, everything else will not be. The only difference between **source** and **control** is that you created the **source** paths, while ArX created the **control** paths. For files that are not being archived, ArX has to know whether it should warn the user during `tree-lint`. Only **unrecognized** files trigger warnings just by being classified as **unrecognized**. This is summarized in Table 5.1.

The algorithm that ArX uses to classify a path is:

1. If the path is a directory and has an `_arx` subdirectory, then it is a **nested_tree**.
2. If the path is in the `_arx` directory, then it is **control**.
3. If the path has an inventory id, then it is **source**.
4. If the path's name matches with the regex for **ignore**, then it is **ignore**.
5. Otherwise, it is **unrecognized**.

"`arx inventory`" will print out a list of all of the paths and how they have been classified. By default, `inventory` will not print out the **control** paths. The default regex for **ignore** is empty. You can change it with "`arx ignore`". For example, to change the ignore regex to ignore files ending with `.o`, `.bak`, or `~`, the command would be

```
arx ignore "^.*(.o|.bak|~)$"
```

ArX uses Boost.Regex, which uses the regular expression syntax described in ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects (FWD.1).

5.14 Pristine Trees

ArX normally stores a complete copy of the project tree in the `_arx` directory. This allows commands which need to compare against a previous revision, such as `commit`, `diff`, `undo`, and `file-undo`, to complete quickly. Also, if ArX has to get a particular revision, it can use that pristine tree as a base to start from instead of having to fetch everything from the archive.

Usually, a project tree will only have the pristine tree of the latest revision. ArX will automatically keep it up to date for you. Sometimes, you may find it useful to have pristine trees from other revisions, such as revisions that have branched off of yours. You can query, add, or remove pristine trees with `tree-cache`.

One problem with pristine trees is that they do take up more space. See section 5.9.2 for one strategy for ameliorating that.

5.15 Additional Tools

Included with the ArX distribution are a few additional tools.

- A bash completion code to make typing many of the commands less onerous.
- An emacs mode which integrates ArX into the editor.
- A python script `check_moved.py` which is useful when importing patches from non-ArX users. A diff can simulate a file rename by deleting and re-adding the file. `check_moved.py` will detect that and fix up the internals of ArX to correctly track that move.
- A patch-queue manager `pqm`. See Section 5.3.4.

Chapter 6

Beyond this manual

This manual has presented most of the commands available. If you want to find out what all of the command's are,

```
arx --help-commands
```

will print them out. By necessity, this manual has not plumbed all of the various options to the commands. All of the commands have a help screen that can be accessed with the –help command.

Appendix A

Patch Algorithm

There are three possible relations between two objects: parent (p), child (c), and other (o). A parent is a parent directory, child is a child directory, and other is something that is not in the same hierarchy. Viewed this way, there are nine different possible ways to move things:

1. p->p
2. p->c
3. p->o
4. c->p
5. c->c
6. c->o
7. o->p
8. o->c
9. o->o

We also want things that are in a directory that is being renamed or deleted to be automatically renamed or deleted if they are not otherwise specified.

If someone doesn't want things to be automatically deleted, then we can only delete directories that are already empty, because the contents have all been eliminated.

The basic algorithm is:

1. Get a list of all renames and deletes, and sort it so that the bottom-most elements are first. That is, if we have
   `a/`
   `a/b/`
   `a/c/`
   `a/d/`
   `a/b/c`

`a/c/c`
Then it should get sorted as something like
`a/d/`
`a/c/c`
`a/c`
`a/b/c`
`a/b`
`a/`
Note that this is both renames and deletes. For example, `a/c` might be deleted and `a/c/c` might be renamed.

2. In this bottom-up ordering, we rename the deleted files to `,,delete-0`, `,,delete-1`, ... and renamed files to `,,renamed-0`, `,,renamed-1`, ... in the temp directory. If a source does not exist, complain and put a note somewhere.

3. Figure out where a path should go. If the path is just being renamed (foo/a -> foo/b) as opposed to being moved (foo/a -> bar/a), then just rename the path regardless of its current parent. If it is just being moved, then move the path, regardless of its current name. Note that ArX knows if a parent directory has been moved and puts the path in the right place. If the destination parent does not exist, signal a conflict and put the path in the destination given by the patch.

4. Sort the destinations of the renames in a top-most fashion (opposite of bottom-up). Move the renames into their destination using this ordering. If the destination exists, rename the destination to (original_name).orig. If _that_ exists, then we try .orig-1, .orig-2, ...

5. If we are removing deletes, then just delete all of the „delete objects. If we are keeping them, then do a similar rename for the `,,delete-*` files, moving things to a `,,removed-by-dopatch` directory but it has the original name. There should be no conflicts when doing this rename.

6. Delete the temporary directory. There should be nothing in it.

7. Apply regular and metadata patches to paths.

Appendix B

Conflicts

There are 12 different types of possible conflicts. Most of these types are related to moving and renaming paths. One thing to keep in mind is that ArX handles renames (`foo/a -> foo/b`) separately from moves (`foo/a -> bar/a`). So there can be conflicts related to the parent directories separately from the renamed path.

1. Merge: There was a conflict when applying a three-way merge to a file. This is the most common type of conflict when merging files, where two people make conflicting changes to a single file. ArX prints out the locations of the partially merged file, the original version in the tree, the ancestor's version, and the sibling's version. For example, ArX might print out

   ```
   foo foo.tree foo.ancestor foo.sibling
   ```

   If a merge script exists (see Appendix C.1), ArX will invoke the script for these files.

2. Patch: There was a conflict when applying a patch to a file. This is the most common type of conflict when using `replay` or the `patch` algorithm in `merge`, where two people make differing changes to the same file. ArX prints out the locations of the file with perhaps some parts of the patch applied, a copy of the file before it was patched, and a copy of the rejected hunks of the patch. For example, it might print out

   ```
   foo foo.orig foo.rej
   ```

   If a patch-merge script exists (see Appendix C.2), then ArX will invoke the script on these files.

3. xdelta: There was a conflict when applying a patch to a binary file. ArX uses the xdelta algorithm to compute diffs between binary files, and patches to binary files only work if the file is exactly what is expected. So there is no fuzz factor to allow for modified files to be patched. ArX prints out the name of the file and the rejected xdelta patch. For example, it might print out

   ```
   foo foo.xdelta
   ```

Unfortunately, there is not much that you can do with `.xdelta` files. They use a different format than the xdelta program.

4. Move Target: The destination of a rename is already occupied. For example, if the patch renames `foo` to `bar`, and `bar` already exists. ArX prints out the contended name and where the original has been moved. In this example, ArX would print out

   ```
   bar bar.orig
   ```

5. Move Parent: The parent of a path that has been renamed has been changed in some incompatible manner. For example, if a patch renames `foo/a` to `bar/a`, but the file is in directory `baz`. ArX prints out the initial placement of the moved path, the patches initial parent directory, and the patches destination parent directory. In this case, ArX would print out

   ```
   baz/a foo => bar
   ```

6. Rename: The name of a path has been changed in some incompatible manner. For example, if a patch renames `foo/a` to `foo/b`, but the file is already named `foo/c`. ArX prints out the initial location of the moved path, the patches initial location of the moved path, and the patches destination of the moved path. In this case, ArX would print out

   ```
   foo/c foo/a => foo/b
   ```

7. Deleted Parent: The parent directory for the destination of a move has been deleted by this patch. For example, suppose the patch moves `foo/a -> bar/a` and deletes the directory `baz`, but `bar` has been moved into a subdirectory of `baz`. ArX prints out the patches initial and final destination. In this case, ArX prints out

   ```
   foo/a => bar/a
   ```

8. No Parent: The parent directory for the destination of a move path has been deleted outside of this patch. For example, if the patch moves `foo/a` to `bar/a`, but `bar` was deleted before the patch was applied. This differs from a Deleted Parent conflict where the parent directory is deleted in the patch itself. ArX prints out the patches initial and final destination. In this case, ArX prints out

   ```
   foo/a => bar/a
   ```

9. Missing Moves: A path that is being moved seems to be missing. ArX will print out the patches initial and destination location, and the path's inventory id. For example, it might print out

   ```
   foo bar 32472534872abd896dc986de22f87de9fef997a97cbd97e9779824234827648d
   ```

10. Missing Patches: A path that is being patch seems to be missing. ArX will print out the patches path location.

11. Add: A path is being added with the same inventory id. For example, you might have a path `foo` with the inventory id `a9de...`, and you are trying to add a path `bar` with the same inventory id. ArX will print the path you are trying to add, the path that conflicts with it, and the inventory id. For this example, ArX will print out

    ```
    bar foo a9de...
    ```

    Note that ArX will only signal a conflict if either the name or the content of the path is different. So if you apply an ArX patch and then immediately reapply it, you should not get any of these kinds of conflicts.

12. Directory Loop: ArX encountered a loop when trying to move a path. This conflict happens when the patch tries to move a directory to its own subdirectory. For example, suppose the patch moves `foo -> bar/foo`, but the tree already has `foo/bar`. If ArX detects a directory loop, ArX will try to move everything back to where it was before. This may cause additional conflicts if some parent directories are deleted. ArX will print out the paths current location and the patch's initial and destination locations. For this example, ArX will print out

    ```
    foo foo -> foo/bar/foo
    foo/bar bar/foo -> bar/foo/bar/foo
    ```

    Note that ArX inferred the move of `foo/bar -> bar/foo/bar/foo`.

Appendix C

Sample Merge Scripts

C.1 Three way merges

ArX looks in ~/.arx/merge3 for an executable merge script. The script is given four arguments

1. The original tree file
2. The ancestor file
3. The sibling file
4. The destination tree file

C.1.1 Meld

```
rm ‘‘$4’’
mv ‘‘$1’’ ‘‘$4’’
meld ‘‘$2’’ ‘‘$4’’ ‘‘$3’’
```

C.1.2 Xxdiff

```
rm ‘‘$4’’
xxdiff --title1 ancestor --title2 tree --title3 sibling -M ‘‘$4’’ --show-merged-pane
```

C.1.3 kdiff3

```
rm ‘‘$4’’
kdiff3 --L1 ancestor --L2 tree --L3 sibling -o ‘‘$4’’ ‘‘$2’’ ‘‘$1’’ ‘‘$3’’
```

C.1.4 gvimdiff

```
rm ‘‘$4’’
mv ‘‘$1’’ ‘‘$4’’
gvimdiff ‘‘$2’’ ‘‘$4’’ ‘‘$3’’
```

C.1.5 X/Emacs

```
rm ‘‘$4’’
emacs --eval ‘‘(ediff-merge-files-with-ancestor \’’$1\’’ \’’$3\’’ \’’$2\’’ nil \’’$4\’’)’’
```

C.2 Patch merges

ArX looks in ~/.arx/patch-merge for an executable merge script. The script is given three arguments

1. The original tree file
2. The .rej file
3. The .orig file

At present, the only tool that works well with .rej files is X/Emacs. The script for X/Emacs is simply

```
emacs $2
```

The following is a recipe from Miles Bader for using emacs

> If you’re using an up-to-date version of emacs (I mean the original GNU Emacs, I’m not sure about xemacs), it should enter diff-mode automatically when you visit the .rej file. From there, there are several useful commands you can use, for instance, putting the cursor in a diff ‘hunk’, and pressing ‘C-c C-c’ will attempt to jump to the corresponding location in the source file; typing ‘C-c C-a’ while in a hunk will try to actually apply the hunk (and will fail if it can’t). Applying a hunk from diff-mode sometimes succeeds where patch failed, though I’m not exactly sure why, as it’s actually more strict about matching the original file (it doesn’t do ‘fuzzy’ application).
>
> So for instance a typical strategy I’ll use is:
>
> (1) Visit the .rej file in emacs; this will automatically be in diff-mode
>
> (2) Make the buffer writable so I can modify the .rej file; this is just my personal style, you don’t have to do this. diff-mode by default makes the buffer read-only, but I like to delete each hunk successfully applied, to make bookkeeping easier for big .rej files.
>
> (3) Use the command ‘M-U’ first, which converts the .rej file into ‘unified diff’ format, which I find easier to read; again this is not necessary though, just something I like (and of course the buffer must be writable from step (2) to do this!).
>
> For each hunk:

(3) Use C-c C-a to try to apply the hunk; if application succeeds, delete the hunk from the .rej file with 'M-d' (.rej buffer must be writable to do this), and go on to next hunk, otherwise:

(4) Use C-c C-c to find the source location – this command will use line numbers as a backup strategy, so it usually gets you at least close – and see if there's some obvious problem where the source file has change from what the patch is expecting.

(5) If there's an obvious difference, say added code in the hunk's context lines, _modify the hunk_ to match the source, making sure any new lines you add to the hunk include appropriate diff line-start characters (' ', '+', '-'). diff-mode will automatically make sure that the hunk line counts etc are kept up-to-date. Of course this requires care, but I find it easier to think about the interaction of changes if I keep the source file unchanged and update the hunk. If the hunk then applies, then delete it and contine as in step (3).

(6) Sometimes diff generates really big hunks, which include many individual changes, and are difficult to think about as a whole. For these, I often use the diff-mode 'C-c C-s' command, to split the current hunk into two smaller hunks at the current line (this only works in unified diff format, for obvious reasons), and then deal with each smaller hunk individually. Sometimes, if you're not sure where the problem in a big hunk is, you can use C-c C-s to do a binary search for the mismatch point (and use emacs' undo command to undo any split that's not useful).

The above might sound a bit complicated, but really it's not to bad once you know the diff-mode commands.

The crucial thing I think, is that it's _much_ easier to handle non-trivial conflicts with proper .rej files, compared to CVS conflict markers. the main reason I think, is that patch is more conservative, and requires a certain amount of surrounding context to match for a patch to be applied, and includes the failing context in the .rej files so you can see what happened. Together with diff's habit of merging adjacent hunks into bigger hunks, this means that potentially problematic merges are more likely to simply fail – which is a _good_ thing...

CVS requires _no context_, and though this can be convenient for 'obvious' cases, by the time that you realize something is non-obvious, it's already too late, CVS has already applied a bunch of possibly incorrect changes, intermixed with non-applied changes using context markers.

-Miles

Glossary

archive A directory where revisions are stored. See Section 5.1

branch A specific line of development. See Section 5.1.1

inventory id A unique name for a path that persists across renames. See Section 5.13.1

path A file or directory. In many case, ArX treats files and directories in very similar ways.

project A collection of all of the various branches and revisions that make up a particular work.

project tree A directory that contains a working copy of your work.

pristine A protected, unaltered copy of a particular revision, normally stored in a project tree in the {arch} subdirectory. See Section 5.14

revision A snapshot of the work at a particular time, complete with a patch log describing how it differs from previous revisions. See Section 5.1.1

whole-tree commits A commit that involves all of the files in a project tree.