Freie Universität Berlin, Fachbereich Mathematik und Informatik,
Institut für Informatik, Studiengang Informatik Bachelor

BSc Thesis – Investigation of tree conflict handling in selected version control systems

by

Stefan Sperling
sperling@inf.fu-berlin.de
stsp@stsp.name

Mentor: Martin Gruhn
Supervising Professor: Prof. Dr. Lutz Prechelt

Full text in PDF format and LaTeX source files available at
http://www.inf.fu-berlin.de/w/SE/ThesisTreeConflicts

September 2, 2008

Virtually any version control system that facilitates concurrent development via the *Copy-Modify-Merge model* tries to help its users deal with situations where two developers make conflicting changes to the code base.

To help users deal with conflicting changes which need to be merged, conflict detection and resolution mechanisms must be provided by the version control system.

I will describe conflicts which can occur at the level of the directory hierarchy of a software project, and examine to what degree selected general-purpose version control systems in common use today are able to detect such conflicts.

Contents

List of Tables

1 Introduction

Software configuration management (SCM) is the discipline of managing the evolution of large and complex software systems[28]. An important kind of tool used in this discipline is a *version control system*. Such a system records all changes made to a software project in a database often called a *repository*. This facilitates retrieval of arbitrary versions of the project, as well as recording of meta data such as who has made a change, and why.

1.1 Background

Many version control systems facilitate parallel development in a software project via *branching* of development history.

When working on a project with multiple branches of development, the amount of development done in parallel increases with the number of branches. The more development done in parallel, the more likely the occurrence of conflicts when different lines of development are reconciled.

When doing parallel development, developers may have to merge changes made on one branch to another. Some common merging use cases are:

- Merging a bug fix from the *main line* of development (a.k.a. the *trunk*) to a *maintenance branch* (a.k.a. *stable release branch*).
- Merging during maintenance of a *feature branch*, which is forked off the main line to implement a new feature over a longer period of time.

  A feature branch minimises the risk of disturbing other on-going development on the main line. The feature branch will need to be synced with the main line periodically by merging changes made on the main line into it. It will eventually be merged back into the main line entirely, reconciling the two parallel lines of development history.
- Merging between long-lived branches which proliferate during a project's history. This is common in scenarios where software is adapted and maintained over many years to serve the needs of specific users of the software. Branches are created frequently, but are rarely reconciled. Some changes need to be merged between branches, while some are branch-specific and should not be merged to any other branch.

1.2 Motivation

Several approaches to merging changes made in parallel to a software project are known. Mens gives very nice overview in [25], which also suggests that one way of categorising merge approaches is by their representation of software artifacts:

- Textual merging: Treats software as a collection of text files and provides granularity at the level of individual lines or characters in a file. If two developers change the same line in a file, the version control system will ask them to manually determine what the correct content of the line should be.

  For example, CVS[18] operates this way.

- Syntactic merging: Understands the syntax of programs (e.g. by using a parse tree) and tries to avoid producing syntactically incorrect merge results which can occur when using textual merging. This ensures that automatically merged programs will always compile, but they will not necessarily behave correctly at run-time.

  This approach is also useful for merging structured data objects and not just program fragments. For example, Lindholm employs syntactic merging techniques to merge XML documents[23].

- Semantic merging: Attempts to take the meaning of programs (e.g. the relation between a subroutine's input and output values) into account, and tries to preserve the program's behaviour which can be changed inadvertently during syntactic merging. A correct merge result from a semantic merge exposes neither syntax nor run-time errors.

  A language-independent semantic merging model is presented by Berzins in [19].

- Structural merging: Tries to tell purely structural changes, which do not affect the semantics of a program, apart from semantic changes. It does so in order to recognise cases where differently structured program sections are actually semantically equivalent (e.g. after behaviour-preserving refactoring).

Mens further distinguishes between merge algorithms which are state-based, changed-based and operation-based:

- State-based merging only takes differences between two versions of a software fragment into account (two-way merge). It does not take into account how these changes came about.

- Change-based merging takes into account some of the history of how changes came about.

- Operation-based merging is a particular flavor of change-based merging that models changes as explicit operations (or transformations)[25][24].

During merges, it is entirely possible that the source branch and the target branch involved in the merge do not share the same directory tree structure.

For example, in large software projects maintained and enhanced over a decade or more, many items in the project's source tree might eventually be moved to different directories.

Over time, the location of related items in the source tree may start to differ between development branches.

If not handled well by the version control system, the burden of handling these kinds of conflicts lies with developers. In software projects which make heavy use of techniques such as refactoring, conflicts at the level of directory hierarchy can become a serious problem. Refactoring involves program transformations which improve the internal design of a program without changing the program's behaviour[25]. Refactoring may involve renaming of files and directories, which are used to represent classes or packages in object-oriented languages such as Java[1] and Python[2]. This work will take a look at a particular side-effect of structural merge conflicts.

The kind of conflict under consideration does not manifest itself at the level of file content, so file content or even file content semantics are of no concern. Instead, the conflict manifests itself in the structure of the directory tree of a software project.

We will look at cases where, for any reason, conflicting changes made to the directory structure of a software project clash during a merge. Such conflicts will be called **tree conflicts**.

1.3 Approach

I was first faced with tree conflicts while doing development for the Subversion[3] project. Subversion users were reporting great difficulties when merging changes between long-lived branches, and presented the Subversion community with 3 update and 3 merge use cases which Subversion failed to handle in a useful way[27].

The need for a generic approach to tree conflicts became increasingly apparent while working on making Subversion handle the 6 presented use cases correctly. I picked the topic of my thesis to become more familiar with tree conflicts in general, and to illustrate the current state of tree conflict handling in Subversion and other version control systems.

At some point during development discussion, Julian Foad and Nico Schellingerhout presented a matrix plotting operations carried out during a merge against possible states of objects in the target working copy of the merge[21].

This matrix provided the basic idea for an approach to modelling tree conflicts – that it should be possible to derive tree conflict scenarios from the operations offered by a version control system.

In order to examine the behaviour of several version control systems when faced with tree conflicts, it is useful to express merging use cases which involve tree conflicts in an implementation-independent manner. This work describes a model which can be used to derive abstract merging uses cases from the basic mechanisms version control systems offer. Those cases which involve tree conflicts can then be replicated with any version control system.

The behaviour of individual version control systems was tested by writing shell scripts which execute commands of the version control systems in such a way that the tree conflict scenarios described by the model were reproduced.

Because of the complex nature of the problem, an agile and iterative approach had to be taken. The model described in section 2 was continuously improved while working different version control systems through tree conflict use cases expressed by the model.

Additionally, all iterations of the model were repeatedly discussed with people[1], familiar with version control in general and the tree conflict problem in particular which lead to improvements every single time.

The first iterations were able to express many more merge and tree conflict scenarios than the final iteration, but also had inconsistencies which would not be discovered until much later. The large number of expressible cases also meant more work had to be done writing reproduction scripts. Finding a trade-off between a manageable chunk of use cases and the best possible amount of problem space coverage was not easy. Twice, the changes made to the model obsoleted virtually all results I had collected so far, forcing me to restart my investigation of version control tools nearly from scratch.

1.4 Outline

The outline of the remaining sections is as follows:

- Section 2 describes a model which can be used to derive merge use cases which involve tree conflicts.
- Section 3 introduces the version control systems which were examined for their behaviour when faced with tree conflict use cases derived from the model described in section 2.
- Section 4 summarises the behaviour observed.
- Section 5 lists the behaviour observed in detail.
- Section 6 describes possible improvements which could be made to the model described in section 2.
- Section 7 lists people who helped me with my thesis. It also provides a non-exhaustive list of additional version control systems which would be interesting to examine with respect to tree conflicts.

[1] See section 7

2 Modelling Tree Conflicts

One possible approach to modelling of tree conflicts is based on the idea that it should be possible to identify tree conflict scenarios by deriving them from the basic mechanisms version control systems offer. This way, a finite number of possible tree conflict scenarios can be derived, because the number of mechanisms version control systems offer is also finite.

2.1 Tree Conflicts Model

The model presented here is meant to be independent of the implementation of any particular version control system. Its goal is to *informally describe a sufficient number of tree conflict cases* to investigate, and to provide a rough idea about the requirements a version control system must meet in order to be able to handle tree conflicts gracefully at a fundamental level.

It is *not* the goal of this model to describe *any* possible tree conflict scenario version control system developers and users may face in reality. Version control systems may provide functionality which cannot be represented in this model. Users of version control systems may therefore encounter tree conflicts which cannot be represented in this model.

Since the model is purely meant to be used as a descriptive aid, it is not formally correct. Many details and concepts which would be necessary to use the model for formal tasks, such as describing a merging algorithm, have intentionally been omitted.

The model is ID-based, which means that it makes use of unique identifiers to track the identity of hypothetical objects under version control.

It also makes partial use of the idea of operation-based merging[24], in the sense that a merge knows which operations a change is composed of. It can replay these operations in order, one-by-one, to produce the merge result. This is useful because tree conflicts are easier to describe as the result of conflicting operations rather than a conflict between the state of two different trees.

But the model does not employ operation-based merging techniques in order to resolve conflicts automatically. Conflict resolution is outside the scope of the model.

The model assumes common ancestry between the source and the target of a merge. The problem of merging unrelated lines of history is outside the scope of this model.

Collectively, these properties make the model suitable to describe a small set of tree conflicts, but also make it unlike many of the general-purpose version control system in common use today. Many systems use state-based instead of operation-based merging, and many do not have a concept of object IDs.

2.1.1 Merge Scenarios

Def. 1 *A* **merge scenario** *consists of:*

- *Two revisions of a source* **tree**.
- *A target tree.*
- *An* **object** *in the source tree, with a particular ID.*
- *An* **operation**, *which was carried out on the object in the source tree between the two revisions of the source tree. The operation shall be replayed on a corresponding object in the target tree.*
- *A set of* **flags**, *associated with the object in the target tree.*

All highlighted terms will be defined further below. It may help to come back here and read this definition again once you are familiar with these terms.

Which scenarios actually involve a tree conflict is to some degree up to interpretation. For example, a new file created on top of an existing file could be regarded as either a possible text conflict, or a possible tree conflict. Valid points for both sides of the argument can be made. The desired merge result may depend on semantics and policies which are determined by the developers or users of a version control system.

Nevertheless, we can define a rule of thumb to help us identify which merge scenarios cause tree conflicts:

Def. 2 *If an operation to be replayed in the target line of history conflicts with another operation already carried out in the target line of history in such a way that the directory tree structure resulting from the merge cannot be unambiguously determined, the merge scenario is a* **tree conflict scenario**.

In particular, this includes any combination of conflicting operations that leads to an object being lost, or that leads to modifications made to the content of an object being lost.

Virtually any scenario where the types of objects involved are incompatible is a tree conflict. It should be obvious that it does not make sense to try to merge a directory with a file. When discussing scenarios, we will always assume that the objects involved are compatible.

2.1.2 Objects

Def. 3 *An* **object** *is a file or directory under version control.*

When an object is first created, it is assigned a **unique identifier** (**ID**) which does not change during the life time of the object.

If an object is part of a tree conflict scenario, it is said to be a **victim** of a tree conflict.

Objects have content, which can be modified. Changing the content of an object does not change the object's ID. File content is usually text or binary data. This model never looks at file content. Directory content describes the objects reachable via the directory, and is used to build up a tree of objects (see below).

2.1.3 Trees

Def. 4 *A* **tree** *is a structure defined by a set of objects.*

A tree has a single object at its root, which is always a directory. We refer to the tree's root object as the **root directory**.

Each directory contains a **directory entry** for every object reachable through it. Each directory entry maps an arbitrary name to an object with a certain ID. An object can thus be reachable by a **path**, which contains the names of the directory entries needed to be traversed to reach the object, from the root directory onwards. This is how a tree structure is formed by objects.

There may only be a single directory entry occupied by an object. Multiple directory entries pointing to the same object are not allowed in this model.

The model is capable of identifying different **revisions** of trees. Each such revision represents a snapshot of the tree. Each revision ties together a set of objects with the content they had at the point in time the revision was created.

Creating a new **line of history** creates a new tree with the same content as the original one. This implies that object IDs are not changed when a new line of history is created. Different lines of history can be modified independently[2]

2.1.4 Merging

Def. 5 *The operations which caused the changes between any two revisions of trees are collectively referred to as a* **changeset**. **Merging** *refers to the process of taking such a changeset which was created in a tree representing one line of history (the* **source tree***), and applying it to another tree representing a different line of history (the* **target tree***).*

[2] A "line of history" is also known as a "branch" in version control jargon, but the use of this term can be confusing because it denotes a branch of a project's development history rather than a branch (in the sense of "subtree") inside the versioned tree itself.

The model assumes that both lines of history are **ancestrally related**, that is, the lines of history of the source and target tree meet somewhere in the past.

Conceptually, a merge operates on three trees, namely the target tree, which it modifies, and two revisions of the source tree[3]. The two revisions of the source tree determine the changeset to be applied to the target tree. The changeset is composed of the series of operations which transform the source tree at the **merge-left revision** into the source tree at the **merge-right revision**.

No single operation which is part of the changeset to be merged may be omitted. Because the model assumes common ancestry, this means that the changeset must represent a contiguous range of revisions of the source tree.

Figure 1 illustrates the relationship between the lines of history and their revisions involved in a merge.

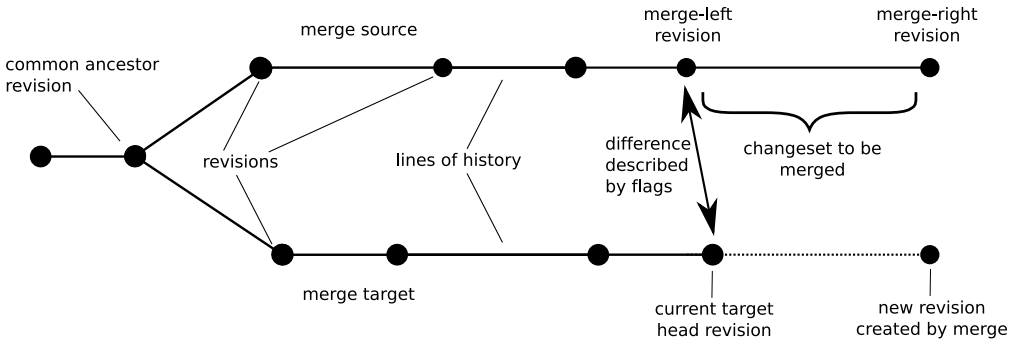

Figure 1: Lines of history and revisions involved in a merge

The lines shown in the figure represent lines of history. The branching point in the figure, at the common ancestor revision, represents the event when history was split into two independent lines of history. Since that point, the source and target trees have evolved independently.

Each dot on a line of history represents a revision in which arbitrary changes were recorded to the tree. The operations which brought about these changes are not shown.

Note that the changeset to be merged may span multiple revisions, even though in the figure it spans only a single revision.

[3]In practice, the target tree of a merge is usually a working copy, and the source trees of the merge are usually located at two distinct revisions in the repository.

2.1.5 Operations

Def. 6 *An* **operation** *is a way by which objects can be created, modified, or destroyed.*

We can define the following possible operations on objects:

- **Creating** a new object with a new ID at some path in the tree. There may be no other object at the same path in the tree already. If a different object is created at a path after another one was destroyed at the same path, the destroyed object can be considered "replaced".

  This is one of two ways to create an object with a new ID – the other is to create a copy of an object.

  When the creation of an object is merged from one line of history to another, the creation of the object is replayed, but the ID of the object in the target line of history will *not* differ from the ID of the corresponding object in the source line of history. Otherwise, ancestry information would be lost.

- **Modifying** file content.

  Modifications made to a file change the file's content. While this model does not actually concern itself with the nature of file content, the fact that content modification has taken place is still relevant to tree conflicts.

  Modifications to directories cannot be made via this operation. Instead, directories are always modified indirectly via one of the other operations. Modification of a directory means any changes made to the directory's entries (e.g. by creating or destroying files or subdirectories), as well as any modification of a file located somewhere in the subtree rooted at the directory under consideration.

  The latter means that modifications trickle up the directory tree. The root directory of a tree will always be considered modified if any modification was made anywhere in the tree.

- **Destroying** an object. This is the opposite of creating one. Destroying an object does *not* destroy the object's history. The permanent removal of all information about an object, including removal of the object's history, is not possible nor necessary in this model.

- **Copying** an object to another path in the tree. There may be no other object at the path already. Copying an object creates an object with identical content but with a different ID, and at a different path.

- **Moving** an object to a different path in the tree. The object is moved to a different path but retains its ID.

2.1.6 Flags

Def. 7 *The set of* **flags** *associated with an object in the target tree of a merge describe those operations carried out on the object which have not been carried out on the corresponding object at the merge-left revision of the source tree.*

Flags can be divided into the following categories:

- A flag signalling that the object was created: **created**
- Flags signalling changes made to the object (either to object content or to the object's location in the tree): **modified, moved**
- A flags signalling that the object was destroyed: **destroyed**

Flags from different categories cannot be set simultaneously.

Together, the flags of all objects in the target tree describe all operations needed to transform the source tree at the merge-left revision into the target tree.

If a flag cannot be represented in a version control system, the system will probably be unable to deal with tree conflicts involving the flag.

The meaning of all flags will now be described in detail:

- The **created** flag signals that the object is new, either because it has been copied or newly created.

  This flag may also signal the presence of an object in the target tree which was destroyed in the source tree before the merge-left revision. In this case, it indicates that deriving the target tree from the source tree includes undoing the destruction of that object.

  In either case, the created flag implies that there cannot be an object with the same ID in the source tree at the merge-left revision. There may still be an object with a different ID at the same path, however.

- The **modified** flag indicates that the object's content has been modified.

  For files, this means file content modification.

  For directories, this means that any object in the subtree rooted at the directory was changed. If a directory does have this flag set, we know that the subtree rooted at the directory was modified. Conversely, if a directory does not have this flag set, we know that the subtree rooted at the directory is unmodified.

- The **moved** flag signals that the path the object occupies has changed with respect to the object with the same ID in the source tree at the merge-left revision.

- The **destroyed** flag signals that the object is not present in the target tree anymore. It implies that the object did exist in some past revision of the target tree.

2.2 An example Merge Scenario

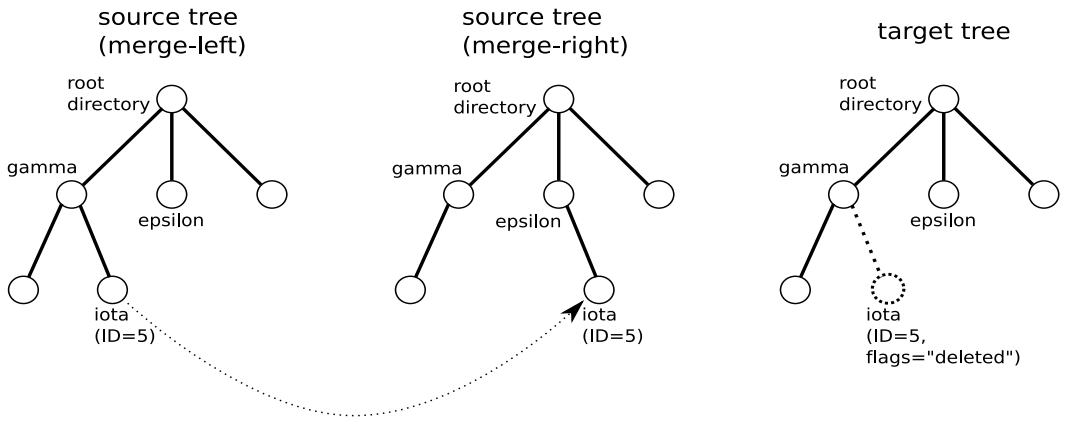

Figure 2: An example merge scenario. A file was renamed in the source tree, but deleted in the target tree. The rename operation cannot be replayed in the target tree, so this is a tree conflict.

Each operation in the changeset applied to the target tree results in a merge scenario.

Figure 2 shows an example scenario where the changeset to be merged consists of the move of a file. The same file has been deleted in the target tree's line of history. The file's deletion is indicated by the "deleted" flag which is set on the file.

This merge scenario happens to be a tree conflict scenario – the move cannot successfully be replayed in the target tree because the file to be moved has been deleted.

2.3 Discussion of Merge Scenarios

Tables 1 and 2 summarise all merge scenarios which can be expressed in the model, and indicate which of them are tree conflict scenarios.

We do not need to distinguish between files and directories when describing these scenarios, with a few exceptions:

- The "Modify" operation only applies to files.
- When a "modified" flag is set on a file, only one object is affected (the file itself).

  But when a "modified" flag is set on a directory, one or more operations were carried out anywhere in the subtree rooted at the directory. In other words, one

		Operation			
	Object state	1) Modify	3) Destroy	4) Copy (Phase 1)	7) Move (Phase 1)
a)	object does not exist	**c**	m	**c**	**c**
b)	no flags set	m	m	2	2
c)	modified flag set	x	**c**	2	2
d)	created flag set	-	-	-	-
e)	destroyed flag set	**c**	m	**c**	**c**
f)	moved flag set	m	**c**	2	**p**

Table 1: Merge scenarios where operations look up objects by ID

Legend:

-	Cannot happen	2	Proceed to phase 2
/	Irrelevant	**c**	Always a tree conflict
m	Merge succeeds	**p**	Tree conflict if destination paths differ
x	Possible text conflict		

or more flags are set on some object inside the subtree rooted at the directory.

With this in mind, the scenarios are valid for either files or directories. So when reading their descriptions, you can mentally replace the word "object" with either "file" or "directory". In the remainder of this section, the scenarios shown in Tables 1 and 2 will be discussed in detail.

1. Operation: Modify

   In this case, we try to apply a content modification to a file (recall that directories cannot be modified by applying a modify operation to them, they can only be modified indirectly).

   First, we look up in the source tree the object ID of the file the modification was made to. Then, we look for a file with the same ID in the target tree.

   Next, any of the following can happen:

   1a) Object does not exist – We cannot find a file with a matching ID at all. This means that the file has never existed in the target tree. Since it exists in the source tree, it must have been independently created in the source tree's line of history. The revision range of this merge is too narrow, because the merge could only succeed if the revision of the source tree which created the file was included in the revision range. Since there is no way we can apply the modification, **this case is a tree conflict**.

   1b) No flags set – The file exists and has none of its flags set. It can safely be modified. There will be no conflict.

		Operation		
	Object state	2) Create	5) Copy (Phase 2)	7) Move (Phase 2)
a)	object does not exist	m		
b)	no flags set	-		
c)	modified flag set	/		
d)	created flag set	**c**		
e)	destroyed flag set	m		
f)	moved flag set	**c**		

Table 2: Merge Scenarios where operations look up objects by path

Legend:

-	Cannot happen	m	Merge succeeds
/	Irrelevant	**c**	Always a tree conflict

1c) Flag: modified – The file has been modified in the target tree relative to the source tree's merge-left revision. The modification can be merged into the file in the target tree, with the possibility of a text conflict.

1d) Flag: created – This cannot happen. It means that we found a newly created file in the target tree with the same ID as the file in the source tree. Because newly created files always get assigned unique new IDs, the target tree cannot carry a newly created file with the same ID as any of the files in the source tree.

1e) Flag: destroyed – This means that the object has been destroyed in the target tree relative to the object with the same ID in the merge-left revision of the source tree. We cannot apply the modification, so **this case is a tree conflict**.

1f) Flag: moved – The file exists but occupies a different path in the target tree than it does in the source tree at the merge-left revision. Our model is smart enough to simply apply the modification to the file at its new location.

2. Operation: Create

   In this case, a new object was created in the source tree, and we want to replay the creation of the object in the target tree. In order not to lose ancestry information, the ID of the newly created object on the target tree should match the ID of the object in the source tree. It is guaranteed that there is no object with the same ID in the target tree already.

   First, we look up in the source tree the path of the object which was created. Then, we search the target tree for an object at the same path[4]

[4]Note that the last directory component of the path is simply assumed to exist, and that scenarios involving renamed directories somewhere along the path cannot be expressed. There was not enough time left before deadline to add the necessary features to express such conditions. See section 6 for more information.

Next, any of the following can happen:

2a) Object does not exist – There is no object at this path. The object can be created in the target tree.

2b) No flags – This cannot happen.

This means that the object we found at this path does not differ, neither in content nor path occupied, from an object with the same ID in the source tree at the merge-left revision.

This implies that an object already existed at this path in the source tree at the merge-left revision. This cannot be the newly created object, because that object was created in the source tree within the revision range defined by the merge-left and merge-right revisions. It cannot have existed already at the merge-left revision in the source tree.

One possibility is that, in the source tree, the newly created object replaced a different object which occupied this path before it. Depending on the revision range to be merged, the merge can either include the destruction of that object or not.

If it does, then this case cannot happen because the merge would destroy the object before trying to create a new one at this path.

If it does not, the object is still present in the target tree despite having been destroyed in the source tree. In this case, it must have its created flag set, because otherwise the flags would not accurately describe the difference between the source tree at merge-left and the target tree.

It could also be that the object was copied to this path from some other path in the target tree. In this case, it must also have its created flag set.

It could also be that the object was moved here from some other path in the target tree. In this case, it must have its moved flag set.

There is no other way an object could already be at this path. So if there already is an object at this path, it is impossible that it does not have any of its flags set.

2c) Flag: modified – This flag is irrelevant in this case. It cannot happen to be the only flag set. See the "No flags" case above for more information.

2d) Flag: created – There already is an object at this path.

This means that two objects have been created independently in the source and in the target tree, both occupying the same path. The object already present in the target tree is guaranteed to have a different ID from the object we are trying to create.

We cannot create the object at this path, so **this case is a tree conflict**[5].

[5]For files, it could be argued that a textual merge of the file to be created and the file already present should be attempted. However, if a merge does try to create a file, we can assume that the file is totally new and has never before been in the target tree. So a textual merge is unlikely to succeed. Also, creating a file is an operation carried out at the tree level, not at the level of file content. So raising a tree conflict is arguably more appropriate here than raising a text conflict.

2e) Flag: destroyed – There was an object at this path in the target tree which has been destroyed relative to the object with the same ID in the merge-left revision of the source tree. This leaves its path unoccupied. So we can safely create a new object at this path.

2f) Flag: moved – Same as "Flag: created" above, except that the object in the target tree was not newly created, but has been moved to this path from another path, relative to the left-merge revision of the source tree. Nevertheless, **this case is a tree conflict**.

3. Operation: Destroy

   In this case, an object was destroyed in the source tree, and we want to replay the destruction of the object in the target tree.

   First, we look up in the source tree the object ID of the object which was destroyed. Then, we look for an object with the same ID in the target tree.

   Next, any of the following can happen:

   3a) Object does not exist – This means that object has not ever existed in the target tree. The destroy action can be transformed into a no-op. The merge result is not ambiguous, so there is no conflict.

   3b) No flags – The object exists in the target tree, and can be destroyed safely, since it has not been changed relative to the merge-left revision of the source tree.

   3c) Flag: modified – This means that the object has been modified in the target tree. It is not safe to destroy the object, because changes made to it in the target tree would be lost. So **this case is a tree conflict**.

   3d) Flag: created – This cannot happen. It means that we found a newly created object in the target tree with the same ID as the object which was destroyed in the source tree. Because newly created objects always get assigned unique new IDs, the target tree cannot carry a newly created object with the same ID as any of the objects in the source tree.

   3e) Flag: destroyed – This means that both the source and the target trees decided that the object should to be destroyed. There is no conflict.

   3f) Flag: moved – This means that the object exists in the target tree, but has been moved to a different path with respect to the object which was destroyed in the source tree. This could mean that the target tree is interested in keeping the object at its new location, so **this case is a tree conflict**, to be on the safe side.

4. Operation: Copy (Phase 1)

   In this case, an object in the source tree was copied to another path. We want to replay the copy operation in the target tree.

We will refer to an object of which a copy was made as the *copy source object*, and to the object which has been created as a copy of the original object as the *copy target object*.

The copy operation needs to be replayed in two phases.

In the first phase, we look up the ID of the copy source object. It will then try to find an object with the same ID in the target tree.

Next, any of the following cases can happen:

4a) The object does not exist – We cannot find an object with a matching ID at all. This means that the copy source object has never existed in the target tree. Since it exists in the source tree, it must have been independently created in the source tree's line of history. The revision range of this merge is too narrow, because the merge could only succeed if the revision of the source tree which created the copy source object was included in the revision range. Since there is no way we can replay the copy operation, **this case is a tree conflict**.

4b) No flags – This means that the copy source object is present in the target tree and can probably be copied to another path. We proceed to copy phase 2.

4c) Flag: modified – This means that the copy source object is present in the target tree, and carries local modifications. Nevertheless, it can probably be copied to another path, preserving its local modifications. We proceed to copy phase 2.

4d) Flag: created – This cannot happen. It means that we found a newly created object in the target tree with the same ID as the copy source object. Because newly created objects always get assigned unique new IDs, the target tree cannot carry a newly created object with the same ID as any of the objects in the source tree.

4e) Flag: destroyed – This means that the object has been destroyed in the target tree relative to the merge-left revision of the source tree. Since the copy source object has been destroyed, it cannot be copied anymore, so **this case is a tree conflict**.

4f) Flag: moved – This means that the copy source object has been moved to another path in the target tree. A copy of the copy source object can possibly still be made, however, so we proceed to copy phase 2.

5. Operation: Copy (Phase 2)

   In copy phase 2, we look up the path of the copy target object in the source tree. We will refer to this path as the *copy target path*. We then try to find an object at the same path in the target tree.

   Next, any of the following can happen:

5a) Object does not exist – There is no object at the copy target path in the target tree, so we can safely replay the copy operation in the target tree.

5b) No flags – This cannot happen. This means that the object we found has no differences with respect to an object with the same ID in the source tree at the merge-left revision. This implies that an object already existed at the copy target path in the source tree at the merge-left revision.

Since the copy operation we want to mirror has created a new object at the copy target path in the source tree, there cannot have already been any object at the copy target path in the source tree at the merge-left revision.

An object with the same ID could exist in the source tree at the merge-left revision only if it occupied a different path than the copy target path. However, this would force the moved flag of the corresponding object in the target tree to be set. So we cannot find such an object without its moved flag set.

5c) Flag: created – This means that two objects have been created independently at this path in the source and in the target tree. They are guaranteed to have a different IDs. We cannot create the copy target object at this path, so **this case is a tree conflict**.

5d) Flag: modified – This flag is irrelevant in this case.

I cannot happen to be the only flag set. For the same reasons as stated for the "No flags" case above, it can occur only in conjunction with the moved flag.

5e) Flag: destroyed – There was an object at the copy target path in the target tree which has been destroyed relative to the merge-left revision of the source tree. Since the object has been destroyed, the copy target path is unoccupied. So we can create the copy target object at the copy target path.

5f) Flag: moved – Same as "Flag: created" case above, except that the object at the copy target path in the target tree was not newly created, but has been moved to the copy target path from another path relative to the left-merge revision of the source tree. Nevertheless, **this case is a tree conflict**.

6. Action: Move (Phase 1)

In this case, an object in the source tree was moved to another path. We want to replay the move operation in the target tree.

We will refer to the path which the object used to occupy before being moved as the *move source path*, and to the path which the object was moved to as the *move target path*,

The move operation needs to be replayed in two phases.

In the first phase, we look up the ID of the object which was moved in the source tree. We then try to find an object with the same ID in the target tree.

Next, any of the following cases can happen:

6a) The object does not exist – We cannot find an object with a matching ID at all. This means that the object has never existed in the target tree. Since it exists in the source tree, it must have been independently created in the source tree's line of history. The revision range of this merge is too narrow, because the merge could only succeed if the revision of the source tree which created the object was included in the revision range. Since there is no way the merge can replay the copy operation, **this case is a tree conflict**.

6b) No flags – This means that the object is present in the target tree and can probably be moved to another path. We proceed to move phase 2.

6c) Flag: modified – This means that the object is present in the target tree, and carries local modifications. Nevertheless, it can probably be moved to another path, preserving its local modifications. We proceed to move phase 2.

6d) Flag: created – This cannot happen. It means that we found a newly created object in the target tree with the same ID as the object which was moved in the source tree. Because newly created objects always get assigned unique new IDs, the target tree cannot carry a newly created object with the same ID as any of the objects in the source tree.

6e) Flag: destroyed – This means that the object has been destroyed in the target tree relative to the merge-left revision of the source tree. Since the object has been destroyed, it cannot be moved to another path anymore, so **this case is a tree conflict**.

6f) Flag: moved – This means that the object has been moved to another path in the target tree.

If we are lucky, the object in the target tree will have been moved to the move target path already, which would transform this move operation into a no-op.

If we are not lucky, then the object has been moved to a different path than the move target path. If so, **this case is a tree conflict**, because the source and target trees have conflicting ideas about where in the tree this object should be.

7. Operation: Move (Phase 2)

In move phase 2, we first try to find an object at the move target path in the target tree.

Next, any of the following can happen:

7a) Object does not exist – There is no object at the move target path in the target tree, so we can safely replay the move operation in the target tree.

7b) No flags – This cannot happen. This means that the object we found has no differences with respect to an object with the same ID in the source tree at the merge-left revision. This implies that an object already existed at the move target path in the source tree at the merge-left revision.

Since the move operation we want to mirror has move the object to the move target path in the source tree, the object cannot have already been at the move target path in the source tree at the merge-left revision.

An object with the same ID could exist in the source tree at the merge-left revision only if it occupied a different path than the move target path. However, this would force the moved flag of the corresponding object in the target tree to be set. So we cannot find such an object without its moved flag set.

7c) Flag: created – This means that two objects have been created independently at this path in the source and in the target tree. They are guaranteed to have a different IDs. We cannot move the object to the move target path, so **this case is a tree conflict**.

7d) Flag: modified – This flag is irrelevant in this case.

I cannot happen to be the only flag set. For the same reasons as stated for the "No flags" case above, it can occur only in conjunction with the moved flag.

7e) Flag: destroyed – There was an object at the move target path in the target tree which has been destroyed relative to the merge-left revision of the source tree. Since the object has been destroyed, the move target path is unoccupied. So we can move the object to the move target path.

7f) Flag: moved – Same as "Flag: created" above, except that the object at the move target path in the target tree was not newly created, but has been moved to the merge target path from another path, relative to the left-merge revision of the source tree. Nevertheless, **this case is a tree conflict**.

3 Introduction to Investigated Version Control Systems

The following sections introduce the version control systems examined in this study. They were chosen from a set of interesting candidates[6].

Note that this introduction is not meant to be exhaustive. If you need more information than presented here, please follow the appropriate links in the References section to search for each version control system's official documentation.

I started out with Mercurial because I was interested to learn more about it. Git was considered because of its implicit rename tracking and strong focus on merging. Finally, Subversion was chosen primarily to illustrate its lack of tree-conflict handling, and because I am very familiar with it.

I would have liked to examine more systems, but did not have enough time.

The systems examined fall into two distinct categories:

[6] Others considered are listed in section 7.2.

1. *Distributed* version control systems.
2. *Centralised* version control systems.

Distributed version control systems differ from centralised version control system in the way the repository is stored[7].

With distributed systems, each developer working on the project stores a private copy (or "clone") of the repository to work on. Every repository represents one or more distinct lines of history of the project. Changes made in one repository can be merged into another to reconcile different lines of history.

With centralised systems, there is only a single[8] repository stored for the entire project. Each developer works in a "working copy" obtained from the same repository. Each working copy is associated with a particular line of development history. Multiple lines of history may exist in the central repository to facilitate parallel development, and changes can be merged between them for history reconciliation[9].

3.1 Introduction to Mercurial

Mercurial[4] is a distributed version control system, released under the GNU General Public Licence.

I have used Mercurial version 1.0.1 for testing.

3.1.1 Mercurial's repository

Mercurial, being a distributed system, implements support for different lines of history via cloning of repositories.

Mercurial's repository is designed in a similar fashion as the repository of the Monotone[22] version control system. Mercurial applies a hash function to each file under version control. Naturally, the hash value of a file changes each time a change is made to it. A snapshot of the project's tree can then be represented as a list of hashes and paths of all the files in the tree. In Mercurial, as in Monotone, such a list is called a "manifest"[22, p.4][26, p.35]. By linking each manifest to its predecessor, snapshots of the trees can be tracked through history.

Since the manifest itself can be stored in a file, a hash of that file's content can be used to identify revisions of the versioned tree. This revision identifier is virtually universally

[7] Proponents of one approach often argue at length with proponents of the other about the various advantages and disadvantages involved. Neither approach fits everyone's way of working.

[8] Some centralised systems, such as Subversion, also support repository mirroring. But copies of the repository are read-only.

[9] Some distributed systems, e.g. Mercurial and Git, also support multiple lines of history inside a single repository.

unique, an important invariant which needs to be maintained to identify revisions when merging changes between different repositories.

3.1.2 Merging in Mercurial

Mercurial avoids some tree conflict scenarios because of its design. It is not possible to specify a revision range to be merged. Rather, a revision up to which should be merged must be specified. The merge-left revision is always the last merged revision, or the common ancestor revision if the branches have not ever been merged yet.

This effectively means that either all changesets applied to the merge source since the branch point are applied to the target branch, or no merge is done at all. This makes some of our tree conflict scenarios impossible[10].

So before a merge tries to modify a file that does not yet exist in the target branch, the file will be created in the target branch by a parent revision[11].

3.1.3 Mercurial's rename equals "copy+destroy"

In order to rename a file, Mercurial performs two distinct steps. First, it copies the file to the new location. Then, it deletes the file at the old location.

A consequence of this is that during a merge, renames are also merged in two independent steps. One step creates the file at its new location, the other destroys the file at the old location. This design seems to have repercussions on tree conflict detection.

3.2 Introduction to Git

Git[6], like Mercurial, is a distributed version control system. It shares some design aspects with it, but also differs in some ways.

I have used Git version 1.5.6.4 for testing.

3.2.1 Git tracks content

Just like Mercurial, Git aims to track file content rather than directory tree structure:

[10] There is, in fact, an extension that allows revisions to be "transplanted"[5] from one branch to another without also applying parent revisions. But I have decided to focus only on features provided by plain Mercurial, without any extensions.

[11] This is not how Mercurial actually implements this case. In fact, the file is simply created with the content it has in the merge-right revision. But conceptually, the behaviour described is what happens.

> Git is a *content tracker*, where content is de facto defined as "whatever is relevant to the state of a typical sourcecode[*sic*] tree". Basically, this is just files' data and "executable" attribute.[7].

To achieve this, Git identifies files by their content's hash value, just like Mercurial does. Similarly, Git uses the hash of the complete content of a revision to universally identify revisions of the versioned tree.

3.2.2 Git has an "index"

Git has the interesting concept of a staging-area, called the "index"[8] (or "cache"), which is logically situated between the repository and the working copy. Commits are not made directly from the working copy to the repository. Rather, changes have to be added to the index before they can be committed. Git makes use of the index during merging, for example by placing full copies of conflicting files into it. A file of which multiple versions exist in the index is said to be "unmerged". The user can then retrieve different versions of files from the index as required while resolving the conflict.

3.2.3 Git detects renames implicitly

To detect renames, git compares content of files which were deleted in a commit with content of files which were added in the same commit. This approach has been claimed to be of great help when dealing with tree conflicts[29].

3.3 Introduction to Subversion

Subversion[3] is a centralised version control system.

I have used Subversion 1.5.1 for testing.

3.3.1 Subversion versions trees

Subversion manages a single versioned directory tree in its repository. Each change committed to the repository creates a new revision of this tree. Revisions are represented as snapshots of the state of the tree in form of a DAG[12]. Only changed files and directories are stored explicitly when a new change to the tree is recorded. A revision refers to files and directories from older revisions in case these files or directories have not been changed in the revision.

[12]Direct Acyclic Graph

Each revision of the tree is identified by an integer number, which increases any time a change is recorded. Since Subversion is a centralised version control system, a revision identifier which is unique to the repository, instead of being universally unique, is sufficient.

3.3.2 Branches are copies in Subversion

Different lines of history are represented by distinct copies of the same subtree. The copies live below one or more arbitrary subdirectories of the versioned tree. This is different to the tree conflicts model described in section 2, where a different line of history is essentially represented as another "dimension" of the same tree, rather than a copy of the tree.

3.3.3 Subversion treats directories as first-class objects

Subversion versions directories explicitly, instead of implicitly like Mercurial and Git do. Directories are first-class versioned objects, just like files.

Although this makes Subversion potentially capable of handling tree conflicts involving directories much better than Mercurial and Git, this is not the case (see section 4).

3.3.4 Subversion's rename equals "copy+destroy"

Like Mercurial, Subversion implements renames a "copy+destroy".

This brings with it the same problems as described for Mercurial in section 3.1.3.

4 Investigation Results Summary

Tables 3 and 4 list the results of the examination and provide a rough comparison of the tree-conflict-related capabilities of the investigated tools.

Note that these tables should not serve as a sole reference when selecting one tool over the other for use in a software project. There are many other factors to consider which are equally important when making such a decision (such as usability, compatibility, performance, etc.).

Whether tree conflict detection was a design goal for any of the systems examined is doubtful. It is clearly visible from the tables that all of them have severe short-comings with respect to tree conflict detection.

The only version control system that managed to detect a majority of tree conflict cases is Git. Nearly every tree conflict with file victims expressible in the tree conflicts model

Operation	**Object state**	Mercurial 1.0.1	Git 1.5.6.4	Subversion 1.5.1
Modify	no object	/	**c**	-
Modify	destroyed	**c**	**c**	-
Create	created	x	x	x
Create	moved	x	**c**	x
Destroy	modified	**c**	**c**	-
Destroy	moved	-	**c**	-
Copy phase 1	no object	/	-	-
Copy phase 1	destroyed	-	-	-
Copy phase 2	created	x	x	x
Copy phase 2	moved	x	**c**	f
Move phase 1	no object	/	**c**	-
Move phase 1	destroyed	-	**c**	-
Move phase 1	moved	**c**	**c**	-
Move phase 2	created	x	**c**	x
Move phase 2	moved	x	**c**	f

Table 3: Tree Conflict handling for files

Legend:

-	No conflict detected	x	Treated as text conflict
c	Tree conflict is detected	/	Does not apply
f	merge fails erroneously		

described in section 2 is detected. The conflicting operations (such as "rename" or "add") are reported to the user.

While Git also outperforms the other programs examined at handling tree conflicts involving directories, Git's handling of these cases can still be considered quite poor (especially when compared to how well Git handles tree conflicts involving files).

Mercurial detects a few of the file cases equally well as Git, and only one of the directory cases. Quite a few tree conflicts involving files are treated as text conflicts.

Subversion performs astonishingly bad at detecting tree conflicts. Not a single tree conflict described by the model is detected. A few cases involving files cause text conflicts to be raised. A bug[9] even prevented some merges from running at all.

4.1 Summary of tree conflict handling observed in Mercurial

Because of the way Mercurial implements merging (see section 3.1.2), it cannot happen that a changeset cannot find the file it is destined for in the merge target. The file will always be created in the merge target by a parent revision of the revision applying the changeset to the file.

These cases cannot happen:

Operation	**Object state**	Mercurial 1.0.1	Git 1.5.6.4	Subversion 1.5.1
Create	created	-	-	-
Create	moved	-	-	-
Destroy	modified	-	-	-
Destroy	moved	-	**c**	-
Copy phase 1	no object	/	-	-
Copy phase 1	destroyed	-	-	-
Copy phase 2	created	-	-	-
Copy phase 2	moved	-	-	f
Move phase 1	no object	/	**c**	-
Move phase 1	destroyed	-	**c**	-
Move phase 1	moved	**c**	**c**	-
Move phase 2	created	-	-	-
Move phase 2	moved	-	-	f

Table 4: Tree Conflict handling for directories

Legend:

-	No conflict detected	x	Treated as text conflict
c	Tree conflict is detected	/	Does not apply
f	merge fails erroneously		

- "Files/Modify/no object" (5.1.1)
- "Files/Copy1/no object" (5.1.7)
- "Files/Move1/no object" (5.1.11)
- "Directories/Copy1/no object" (5.1.20)
- "Directories/Move1/no object" (5.1.24)

Mercurial deals especially well with scenarios involving concurrent destruction and modification of files. If a locally modified file is about to be destroyed by a merge, Mercurial stops what it is doing and prompts the user about how to proceed. This alone should save many people some headaches, as it really helps when dealing with these types of tree conflicts.

See these cases for examples of this behaviour:

- "Files/Modify/destroyed" (5.1.2)
- "Files/Destroy/modified" (5.1.5)

Implementing renames as "copy+destroy" (see section **??**) causes problems in situations where "copy+destroy" does not cleanly translate into "move" anymore when applied to

the merge target. The "copy+destroy" is then often split up into "create" and "destroy", which might end up creating an unwanted item in the target branch.

See these cases for examples of this behaviour:

- "Files/Copy1/destroyed" (5.1.8)
- "Files/Move1/destroyed" (5.1.12)
- "Directories/Destroy/moved" (5.1.19)
- "Directories/Move1/destroyed" (5.1.25)

Mercurial makes an effort to warn users about divergent renames, which is good, but users may not notice the warning. Treating divergent renames as a conflict would probably be a bit safer. Still, a warning is better than nothing, and the conflicts can be considered detected.

See these cases for examples of this behaviour:

- "Files/Move1/moved" (5.1.13)
- "Directories/Move1/moved" (5.1.26)

Mercurial treats some tree conflicts as text conflicts.

See these cases for examples of this behaviour:

- "Files/Create/moved" (5.1.4)
- "Files/Copy2/created" (5.1.9)
- "Files/Copy2/moved" (5.1.10)
- "Files/Move2/created" (5.1.14)
- "Files/Move2/moved" (5.1.15)

Apparently because Mercurial does not treat directories as first-class objects, a lot of tree conflicts involving directories are not detected. Only one scenario is treated as a conflict, but is treated as a conflict involving files:

- "Directories/Move1/moved" (5.1.26)

4.2 Summary of tree conflict handling observed in Git

It seems that file moves and deletions need to always be committed to the repository before a merge is run for conflict detection to work reliably.

See these cases for examples of this behaviour:

- "Files/Modify/destroyed" (5.2.2)
- "Files/Destroy/moved" (5.2.6)
- "Files/Move1/destroyed" (5.2.12)
- "Directories/Destroy/moved" (5.2.19)

Git treats tree conflicts involving the creation of files at the same path (be it by explicit creation or by copying) as a text conflict.

See these cases for examples of this behaviour:

- "Files/Create/created" (5.2.3)
- "Files/Copy2/created" (5.2.9)

Git does not allow merges into modified working copies in cases where files would be overwritten.

See these cases for examples of this behaviour:

- "Files/Create/moved" (5.2.4)
- "Files/Destroy/modified" (5.2.5)
- "Files/Copy2/moved" (5.2.10)
- "Files/Move2/created" (5.2.14)

When a renamed file interferes with the creation of another file at the move target path, copies of both files are put in working copy.

See these cases for examples of this behaviour:

- "Files/Create/moved" (5.2.4)
- "Files/Copy2/moved" (5.2.10)

Making full copies of files available in the working copy seems a bit redundant, however. Because in most cases, full copies of conflicting files can be retrieved from the index, except in one of the cases observed:

- “Files/Destroy/moved” (5.2.6)

Two tree conflicts involving file copying not are detected. These are:

- “Files/Copy1/no object” (5.2.7)
- “Files/Copy1/destroyed” (5.2.8)

Sometimes, content of conflicting directories is unconditionally merged into a single directory in the target tree.

See these cases for examples of this behaviour:

- “Directories/Create/created” (5.2.16)
- “Directories/Create/moved” (5.2.17)
- “Directories/Copy2/created” (5.2.22)
- “Directories/Copy2/moved” (5.2.23)
- “Directories/Move2/created” (5.2.27)
- “Directories/Move2/moved” (5.2.28)

In one case, overlapping content of conflicting directories is unconditionally deleted from the target branch:

- “Directories/Destroy/modified” (5.2.18)

In other cases, Git creates directories in the target tree unconditionally:

- “Directories/Copy1/no object” (5.2.20)
- “Directories/Copy1/no object” (5.2.20)

Git detects some tree conflict cases involving directories, albeit indirectly. Only files inside directories are treated as tree conflict victims, not directories themselves. This is not surprising given that Git does not treat directories as first-class versioned objects.

See these cases for examples of this behaviour:

- “Directories/Destroy/moved” (5.2.19)
- “Directories/Move1/no object” (5.2.24)
- “Directories/Move1/moved” (5.2.26)

4.3 Summary of tree conflict handling observed in Subversion

When files are modified, or when files or directories are missing from the merge target working copy, Subversion simply skips them during the merge. Changes made to files or directories skipped during the merge are not applied to the merge target at all.

See these cases for examples of this behaviour:

- "Files/Modify/no object" (5.3.1)
- "Files/Modify/destroyed" (5.3.2)
- "Files/Destroy/modified" (5.3.5)
- "Files/Destroy/moved" (5.3.6)
- "Directories/Destroy/modified" (5.3.18)
- "Directories/Destroy/moved" (5.3.19)

Some tree conflicts are treated as text conflicts by Subversion.

See these cases for examples of this behaviour:

- "Files/Create/created" (5.3.3)
- "Files/Create/moved" (5.3.4)
- "Files/Copy2/created" (5.3.9)
- "Files/Move2/created" (5.3.14)

In some cases, Subversion deletes files or directories in the merge target working copy unconditionally.

See these cases for examples of this behaviour:

- "Files/Destroy/modified" (5.3.5)
- "Directories/Destroy/modified" (5.3.18)

Sometimes copied or moved files or directories are created in the merge target working copy unconditionally.

See these cases for examples of this behaviour:

- "Files/Copy1/no object" (5.3.7)

- “Files/Copy1/destroyed” (5.3.8)
- “Files/Move1/no object” (5.3.11)
- “Files/Move1/destroyed” (5.3.12)
- “Files/Move1/moved” (5.3.13)
- “Directories/Copy1/no object” (5.3.20)
- “Directories/Copy1/destroyed” (5.3.21)
- “Directories/Move1/no object” (5.3.24)
- “Directories/Move1/destroyed” (5.3.25)
- “Directories/Move1/moved” (5.3.26)

Not unlike Git, Subversion sometimes merges content of conflicting directories into a single directory in the merge target working copy.

See these cases for examples of this behaviour:

- “Directories/Create/created” (5.3.16)
- “Directories/Create/moved” (5.3.17)
- “Directories/Copy2/created” (5.3.22)
- “Directories/Move2/created” (5.3.27)

In the following cases, the merge fails entirely due to a bug:

- “Files/Copy2/moved” (5.3.10)
- “Files/Move2/moved” (5.3.15)
- “Directories/Copy2/moved” (5.3.23)
- “Directories/Move2/moved” (5.3.28)

The problem is known to Subversion developers and a fix is being worked on[9].

5 Detailed Results

This section describes the results obtained in detail.

The version control systems examined where tested for their behaviour in each of the tree conflict scenarios described by the model defined in the section "Modelling Tree Conflicts" (2).

All tests were run via custom UNIX Bourne shell scripts[13]. The text of all scripts can be found in Appendix A, and they can also be downloaded from *http://www.inf.fu-berlin.de/w/SE/ThesisTreeConflicts*

The *projtree.sh* script creates a directory tree sufficient to replicate our tree conflict scenarios. This is the directory tree which the script creates:

```
/alpha
/beta
/gamma/delta
/epsilon/zeta
```

Because directories aren't versioned as first-class object in Mercurial and Git, it is not easy to mirror our directory scenarios with them. They cannot carry out some of our actions directly on directories.

For example, in order for directories to appear and disappear we have to create and destroy files within them, respectively.

I opted to only investigate directories with different content because this approximates more closely what happens when developers create or destroy unrelated directories with the same names.

I have tested committing local changes before doing a merge in each of the cases presented.

In case of Mercurial, committing local changes did not affect merge results at all. So it seems that, in general, merge results in Mercurial do not differ depending on whether the working copy is carrying local modifications or not.

For Git and Subversion, the effect of committing local changes before a merge is noted in comments inside the scripts (see Appendix A).

[13] The scripts have been tested on FreeBSD-7.0, OpenBSD-4.3, and Debian GNU/Linux 4.0 (Etch) which uses GNU bash-3.1.17(1)-release as its /bin/sh.

5.1 Detailed results for Mercurial

5.1.1 Files/Modify/no object

This situation cannot happen in Mercurial because of Mercurial's design.

It is not possible to merge a revision without also merging all of its parent revisions that are not yet present in the target branch of the merge. See section 3.1 for details.

5.1.2 Files/Modify/destroyed

Mercurial detects the tree conflict and asks the user how to proceed.

After running *hg-files-modify-destroyed.sh* (see 8.70), the following is printed:

```
remote changed alpha which local deleted
use (c)hanged version or leave (d)eleted?
```

5.1.3 Files/Create/created

Mercurial treats this scenario as a text conflict.

After running *hg-files-create-created.sh* (see 8.84), Mercurial warns us about conflicts:

```
merging eta
warning: conflicts during merge.
merging eta failed!
```

And indeed, the file eta, which was created in the source branch, is merged with the file eta in the target branch. In the target branch, eta carries local modifications, which trigger a text conflict:

```
$ hg status
M eta
? eta.orig
$ cat eta
<<<<<<< local
eta, really different
=======
eta
>>>>>>> other
```

5.1.4 Files/Create/moved

Mercurial treats this scenario as a text conflict.

After running *hg-files-create-moved.sh* (see 8.63), Mercurial warns us about conflicts:

```
merging eta
warning: conflicts during merge.
merging eta failed!
```

In the source tree, a new file eta has been created. The file alpha in the target tree was renamed to eta. Mercurial attempts to merge the two files textually, which results in a text conflict:

```
$ ls
beta      epsilon/  eta       eta.orig  gamma/
$ hg status
M eta
R alpha
? eta.orig
$ cat eta
<<<<<<< local
alpha
=======
eta
>>>>>>> other
$ cat eta.orig
alpha
```

5.1.5 Files/Destroy/modified

Mercurial detects this tree conflict and asks the user how to resolve it.

After running *hg-files-destroy-modified.sh* (see 8.83), Mercurial asks us how to proceed:

```
local changed alpha which remote deleted
use (c)hanged version or (d)elete?
```

5.1.6 Files/Destroy/moved

Mercurial does not consider this case a conflict.

Because it treats a rename as "copy+destroy", it essentially converts the destroy operation made in the source branch into a no-op when merging into the target branch. After

all, the file alpha has already been "destroyed" in the target branch (actually, alpha was moved elsewhere).

After running *hg-files-destroy-moved.sh* (see 8.79), we can see that the renamed file alpha.moved is left in the working copy at its new location, and the file at the old location stays destroyed.

```
$ hg status
A alpha.moved
$ cat alpha.moved
alpha
$ ls alpha
ls: alpha: No such file or directory
```

An alternative merge result would be to destroy alpha.moved, giving precedence to the destruction made in the source branch.

If a commit is made from the working copy now, in Mercurial-1.0.1 a crash similar to the one in the "Files/Move1/moved" case (see 5.1.13) happens. This crash can be worked around by committing the move before updating.

5.1.7 Files/Copy1/no object

This situation cannot happen in Mercurial because of Mercurial's design.

Before a merge tries to copy a file that does not yet exist in the target branch, the file will be created in the target branch by a parent revision.

See also section 3.1.

5.1.8 Files/Copy1/destroyed

Mercurial does not treat this as a conflict.

A file locally destroyed in the target branch can be resurrected at a different location without warning if the file is copied in the source branch.

After running *hg-files-copy1-destroyed.sh* (see 8.72), we see that the copy source file (alpha) stays destroyed, while its copy has been added to target branch:

```
$ hg status
R alpha
$ ls alpha
ls: alpha: No such file or directory
$ cat alpha.copied
alpha
```

The user may have wanted alpha.copied to be discarded instead.

5.1.9 Files/Copy2/created

Mercurial treats this situation as a text conflict.

The output of *hg-files-copy2-created.sh* (see 8.77) contains:

```
merging eta
warning: conflicts during merge.
merging eta failed!
```

Mercurial tries to textually merge the files which were independently created at the copy target path in either branch, which results in a text conflict:

```
$ ls
alpha     beta      epsilon/  eta       eta.orig  gamma/
$ hg status
M eta
? eta.orig
$ cat eta
<<<<<<< local
eta
=======
alpha
>>>>>>> other
```

5.1.10 Files/Copy2/moved

Mercurial treats this situation as a text conflict.

The output of *hg-files-copy2-moved.sh* (see 8.81) contains:

```
merging alpha.copied
warning: conflicts during merge.
merging alpha.copied failed!
```

Mercurial tries to textually merge the files which both want to occupy the copy target path, which results in a text conflict:

```
$ ls
alpha              alpha.copied.orig  gamma/
alpha.copied       epsilon/
```

```
$ hg status
M alpha.copied
R beta
? alpha.copied.orig
$ cat alpha.copied
<<<<<<< local
beta
=======
alpha
>>>>>>> other
```

5.1.11 Files/Move1/no object

This situation cannot happen in Mercurial because of Mercurial's design.

Before a merge tries to move a file that does not yet exist in the target branch, the file will be created in the target branch by a parent revision.

See also section 3.1.

5.1.12 Files/Move1/destroyed

Mercurial does not treat this situation as a tree conflict.

The file renamed in the source branch is created in the target branch at its new location, and is left destroyed in the target branch at its old location.

After running *hg-files-move1-destroyed.sh* (see 8.71) we see that alpha.moved was created in the target branch:

```
$ ls
alpha.moved  beta         epsilon/     gamma/
$ cat alpha.moved
alpha
$ cat alpha
cat: alpha: No such file or directory
```

The user may prefer a different merge result, however, such as discarding alpha.moved as well as alpha.

5.1.13 Files/Move1/moved

Mercurial detects this tree conflict and prints a warning.

After running *hg-files-move1-moved.sh*, (see 8.75), we get a warning about divergent renames:

```
warning: detected divergent renames of alpha to:
 alpha.moved2
 alpha.moved
```

Both files are present in the resulting working copy:

```
$ hg status
A alpha.moved2
$ ls
alpha.moved   alpha.moved2  beta          epsilon/      gamma
$ cat alpha.moved
alpha
$ cat alpha.moved2
alpha
```

Tyring to commit at this point results in a crash in Mercurial-1.0.1:

```
$ hg commit -m "commit anyway"
** unknown exception encountered, details follow
** report bug details to http://www.selenic.com/mercurial/bts
** or mercurial@selenic.com
** Mercurial Distributed SCM (version 1.0.1)
Traceback (most recent call last):
  File "/usr/local/bin/hg", line 20, in <module>
    mercurial.dispatch.run()
  File "/usr/local/lib/python2.5/site-packages/mercurial/dispatch.py", \
  line 20, in run
    sys.exit(dispatch(sys.argv[1:]))
  File "/usr/local/lib/python2.5/site-packages/mercurial/dispatch.py", \
  line 29, in dispatch
    return _runcatch(u, args)
  File "/usr/local/lib/python2.5/site-packages/mercurial/dispatch.py", \
  line 45, in _runcatch
    return _dispatch(ui, args)
  File "/usr/local/lib/python2.5/site-packages/mercurial/dispatch.py", \
  line 364, in _dispatch
    ret = _runcommand(ui, options, cmd, d)
  File "/usr/local/lib/python2.5/site-packages/mercurial/dispatch.py", \
  line 417, in _runcommand
    return checkargs()
  File "/usr/local/lib/python2.5/site-packages/mercurial/dispatch.py", \
  line 373, in checkargs
    return cmdfunc()
  File "/usr/local/lib/python2.5/site-packages/mercurial/dispatch.py", \
  line 356, in <lambda>
    d = lambda: func(ui, repo, *args, **cmdoptions)
  File "/usr/local/lib/python2.5/site-packages/mercurial/commands.py", \
```

```
  line 557, in commit
    node = cmdutil.commit(ui, repo, commitfunc, pats, opts)
  File "/usr/local/lib/python2.5/site-packages/mercurial/cmdutil.py", \
  line 1179, in commit
    return commitfunc(ui, repo, files, message, match, opts)
  File "/usr/local/lib/python2.5/site-packages/mercurial/commands.py", \
  line 555, in commitfunc
    force_editor=opts.get('force_editor'))
  File "/usr/local/lib/python2.5/site-packages/hgext/mq.py", \
  line 2189, in commit
    return super(mqrepo, self).commit(*args, **opts)
  File "/usr/local/lib/python2.5/site-packages/mercurial/localrepo.py", \
  line 832, in commit
    new[f] = self.filecommit(f, m1, m2, linkrev, trp, changed)
  File "/usr/local/lib/python2.5/site-packages/mercurial/localrepo.py", \
  line 712, in filecommit
    meta["copyrev"] = hex(manifest1[cp])
KeyError: 'alpha'
```

Note that this crash can be worked around by committing the move before updating.

This bug was reported and filed in the Mercurial bug tracker as issue #1175[10]. The problem has been fixed in Mercurial-1.0.2.

5.1.14 Files/Move2/created

Mercurial treats this case as a text conflict.

The output of *hg-files-move2-created.sh* (see 8.73) contains:

```
merging eta
warning: conflicts during merge.
merging eta failed!
```

Mercurial tries to textually merge the files which both want to occupy the move target path, which results in a text conflict:

```
$ ls
beta      epsilon/  eta       eta.orig  gamma/
$ hg status
M eta
? eta.orig
$ cat eta
<<<<<<< local
eta, but different
=======
alpha
>>>>>>> other
```

5.1.15 Files/Move2/moved

Mercurial treats this case as a text conflict.

The output of *hg-files-move2-moved.sh* (see 8.76) contains:

```
merging eta
warning: conflicts during merge.
merging eta failed!
```

Mercurial tries to textually merge the files which both want to occupy the move target path, which results in a text conflict:

```
$ ls
epsilon/  eta       eta.orig  gamma/
$ hg status
M eta
R beta
? eta.orig
$ cat eta
<<<<<<< local
beta
=======
alpha
>>>>>>> other
```

5.1.16 Directories/Create/created

Mercurial does not treat this scenario as a conflict.

Instead, after running *hg-directories-create-created.sh* (see 8.68), we see that Mercurial merges the two independently created directories:

```
$ hg status
A eta/iota
$ ls eta
iota   theta
```

Keeping either the locally created directory or the directory created in the source branch are other possible merge results, but it requires the user to untangle the merge of the two directories manually.

5.1.17 Directories/Create/moved

Mercurial does not treat this scenario as a conflict.

Instead, after running *hg-directories-create-moved.sh* (see 8.69), we see that Mercurial merges the newly created directory with the renamed one:

```
$ hg status
A eta/delta
R gamma/delta
$ ls
alpha    beta     epsilon/ eta/
$ ls eta
delta  theta
```

5.1.18 Directories/Destroy/modified

The way I implemented this scenario, Mercurial does not signal a conflict. Instead, overlapping content of the directory destroyed in the source branch and the directory in the target branch is removed from the target branch.

After running *hg-directories-destroy-modified.sh* (see 8.59), the file gamma/delta has been destroyed in the target branch:

```
$ hg status
A gamma/theta
$ ls gamma/
theta
```

The directory gamma still exists because the script locally created a file in order to modify gamma. Had it made textual modifications to gamma/delta this case would have behaved like "Files/Destroy/modified" (see 5.1.5), and conflicts on locally modified files which the source branch had destroyed would have been signalled.

5.1.19 Directories/Destroy/moved

Mercurial does not treat this as situation as a conflict. Instead the destruction carried out on the source branch is treated as a no-op when applied to the target branch.

After running *hg-directories-destroy-moved.sh* (8.61), we see that gamma/delta is destroyed, and because Mercurial implements a move as "copy+destroy", so that gamma is considered destroyed in the target branch, the destruction is treated as a no-op. gamma.moved stays present in the working copy:

```
$ hg status
A gamma.moved/delta
$ ls
alpha        beta         epsilon/     gamma.moved/
```

```
$ ls gamma.moved/
delta
$ cat gamma.moved/delta
delta
```

Another possible merge result would have been the destruction of gamma.moved. While textual modifications seem to follow moves in Mercurial, moves do not seem to be followed when the merge wants to destroy a moved item.

5.1.20 Directories/Copy1/no object

This situation cannot happen in Mercurial because of Mercurial's design.

Before a merge tries to copy a file (or, by extension, a directory) that does not yet exist in the target branch, the file will be created in the target branch by a parent revision.

See also section 3.1.

5.1.21 Directories/Copy1/destroyed

Mercurial does not treat this as a conflict. The copy made on the source branch is created in the target branch regardless of the fact the copy source on the target branch has been destroyed.

After running *hg-directories-copy1-destroyed.sh* (8.82), we see that the file gamma/delta is treated as destroyed, and the copied directory lives on:

```
$ hg status
R gamma/delta
$ ls
alpha         beta          epsilon/      gamma.copied/
$ ls gamma.copied/
delta
```

Another possible merge result would have been to destroy both gamma and gamma.copied. While this requires manual intervention on part of the user, it is easy to achieve.

5.1.22 Directories/Copy2/created

Mercurial does not treat this as a conflict.

After running *hg-directories-copy2-created.sh* (see 8.62), we see that the content of the copied directory (file delta) is merged with the content of the newly created directory (file eta):

```
$ hg status
A gamma.copied/eta
$ ls
alpha         beta          epsilon/      gamma/        gamma.copied/
$ ls gamma.copied
delta  eta
```

5.1.23 Directories/Copy2/moved

Mercurial does not treat this as a conflict.

After running *hg-directories-copy2-moved.sh* (see 8.65), we see that the content of the copied directory (file delta) is merged with the content of the moved directory (file zeta):

```
$ hg status
A gamma.copied/zeta
R epsilon/zeta
$ ls
alpha         beta          gamma/        gamma.copied/
$ ls gamma.copied
delta  zeta
```

5.1.24 Directories/Move1/no object

This situation cannot happen in Mercurial because of Mercurial's design.

Before a merge tries to move a file or, by extension, a directory, that does not yet exist in the target branch, the file or directory will be created in the target branch by a parent revision.

See also section 3.1.

5.1.25 Directories/Move1/destroyed

Mercurial does not treat this as a conflict.

The renamed directory as it exists on the source branch is created in the target branch unconditionally.

After running *hg-directories-move1-destroyed.sh* (8.67), we can see that the directory gamma.moved is created in the target branch, even though gamma was destroyed in the target branch and therefore technically cannot be moved anymore:

```
$ ls
alpha        beta         epsilon/     gamma.moved/
```

Another desired merge result is discarding gamma.moved. While this requires manual intervention before commit, it is easy to do.

5.1.26 Directories/Move1/moved

Mercurial detects as possible conflict and issues a warning. Still, it unconditionally creates the renamed directory as it appears on the source branch to the target branch.

The divergent renames are detected only at the level of directory content when, in fact, the directories themselves were renamed.

After running *hg-directories-move1-moved.sh* (8.85) we get a warning about the divergent rename:

```
warning: detected divergent renames of gamma/delta to:
 gamma.moved2/delta
 gamma.moved/delta
```

Both directories are created in the target branch's working copy:

```
$ ls
alpha         beta          epsilon/      gamma.moved/  gamma.moved2/
$ hg status
A gamma.moved2/delta
$ ls gamma.moved
delta
$ ls gamma.moved2
delta
```

Other possible merge results require manual intervention.

This case is essentially just like "Files/Move1/moved" (see 5.1.13).

5.1.27 Directories/Move2/created

Mercurial does not treat this case as a conflict.

After running *hg-directories-move2-created.sh* (see 8.78), we see that the content of the moved directory (file delta) is merged with the content of the locally created directory (file eta):

```
$ hg status
A gamma.moved/eta
$ ls
```

```
alpha        beta         epsilon/     gamma.moved/
$ ls gamma.moved
delta  eta
$
```

5.1.28 Directories/Move2/moved

Mercurial does not treat this case as a conflict.

After running *hg-directories-move2-moved.sh* (see 8.60), we see that the content of the directory moved on the source branch (file delta) is merged with the content of the locally moved directory (file zeta):

```
$ hg status
A gamma.moved/zeta
R epsilon/zeta
$ ls
alpha        beta         gamma.moved/
$ ls gamma.moved
delta  zeta
```

5.2 Detailed results for Git

5.2.1 Files/Modify/no object

This scenario can be replicated in git via the "git-cherry-pick" command. When cherry-picking a revision that tries to modify a file which is not present in the target branch git indicates a merge conflict.

Running *git-files-modify-no-object.sh* (see 8.10) results in a conflict being raised:

```
CONFLICT (delete/modify): eta deleted in HEAD and modified \
in fcce051... changed eta. \
Version fcce051... changed eta of eta left in tree.
Automatic cherry-pick failed.  After resolving the conflicts,
mark the corrected paths with 'git add <paths>' or 'git rm <paths>' \
and commit the result.
```

This behaviour is good enough, in spite of the confusingly worded message which claims eta was destroyed even though it has never existed in the target branch.

5.2.2 Files/Modify/destroyed

Git detects a conflict only if the destruction of the file which is modified by the merge has been committed. If the destruction is just staged in the index the modified file is resurrected with its new content and the file's destruction is "unstaged" from the index.

When running *git-files-modify-destroyed.sh* (see 8.3) with the line committing alpha's destruction in repos2 removed, git does a fast-forward merge and alpha's destruction gets unstaged:

```
Updating b626197..75f333b
Fast forward
 alpha |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha    beta     epsilon/ gamma/
$ cat alpha
alpha, modified
```

However, Git correctly detects the tree conflict if the destruction of alpha is committed before doing the merge. The source branch's version of alpha ends up in the working copy, but is marked as "unmerged" in the index (which is to say "not merged yet because of a conflict"). Both versions of the file exist in the index.

```
CONFLICT (delete/modify): alpha deleted in HEAD and modified in \
2cc725a7db86d3f6b05c4060924142aa5ec314d8. \
Version 2cc725a7db86d3f6b05c4060924142aa5ec314d8 of alpha left in tree.
Automatic merge failed; fix conflicts and then commit the result.
$ git status
alpha: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   alpha
#
no changes added to commit (use "git add" and/or "git commit -a")
$ ls
alpha    beta     epsilon/ gamma/
$ cat alpha
alpha, modified
$ git ls-files --unmerged
100644 4a58007052a65fbc2fc3f910f2855f45a4058e74 1       alpha
100644 bdbcaf36d1d757776eed566cac34641de540ee83 3       alpha
```

```
$ git cat-file blob 4a58007052a65fbc2fc3f910f2855f45a4058e74
alpha
$ git cat-file blob bdbcaf36d1d757776eed566cac34641de540ee83
alpha, modified
```

5.2.3 Files/Create/created

Git treats this case as a text conflict.

Git will not allow us to do this merge into a locally modified working copy:

```
error: Entry 'eta' would be overwritten by merge. Cannot merge.
```

So *git-files-create-created.sh* (see 8.18), commits local changes to the target branch before merging. After running the script, git signals a conflict and requests manual resolution:

```
   2a6bd66..fcf91ab  master     -> origin/master
Auto-merged eta
CONFLICT (add/add): Merge conflict in eta
Automatic merge failed; fix conflicts and then commit the result.
$ git diff
diff --cc eta
index 780f7e4,1120d0d..0000000
--- a/eta
+++ b/eta
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD:eta
 +eta, but different
++=======
+ eta
++>>>>>>> fcf91abb59864b4a4ba3963e05943fd8db85a6b3:eta
```

5.2.4 Files/Create/moved

Git detects the tree conflict.

Git will not allow us to do this merge into a locally modified working copy:

```
error: Entry 'eta' would be overwritten by merge. Cannot merge
```

So *git-files-create-moved.sh* (see 8.24), commits local changes to the target branch before merging.

Git notes that the renaming of alpha to eta in the target branch clashes with the creation of eta in the source branch, and reports a conflict:

```
   d019013..bdd40ec  master     -> origin/master
CONFLICT (rename/add): Renamed alpha->eta in HEAD. \
eta added in bdd40ec2b3dbe3c5c90289d9ed8ef86cd1d434c6
Added as eta~bdd40ec2b3dbe3c5c90289d9ed8ef86cd1d434c6 instead
Automatic merge failed; fix conflicts and then commit the result.
```

It also makes both files available in the working copy so the user can easily resolve the conflict:

```
$ ls
beta
epsilon/
eta
eta~bdd40ec2b3dbe3c5c90289d9ed8ef86cd1d434c6
gamma/
$ git status
eta: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   eta
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       eta~bdd40ec2b3dbe3c5c90289d9ed8ef86cd1d434c6
no changes added to commit (use "git add" and/or "git commit -a")
$ cat eta
alpha
$ cat eta\~bdd40ec2b3dbe3c5c90289d9ed8ef86cd1d434c6
eta
```

Both files can also be seen in the index:

```
$ git ls-files --unmerged
100644 4a58007052a65fbc2fc3f910f2855f45a4058e74 2       eta
100644 1120d0dcc8dbe13c5c7f9120596bbbf2c651c564 3       eta
$ git cat-file blob 4a58007052a65fbc2fc3f910f2855f45a4058e74
alpha
$ git cat-file blob 1120d0dcc8dbe13c5c7f9120596bbbf2c651c564
eta
```

5.2.5 Files/Destroy/modified

Git detects the tree conflict.

Git will not allow us to do this merge into a locally modified working copy:

```
error: Entry 'alpha' would be overwritten by merge. Cannot merge
```

So *git-files-destroy-modified.sh* (see 8.17), commits local changes to the target branch before merging.

Git notes that the destruction of alpha clashes with the modifications made to it on the target branch, and reports a conflict:

```
   dd31333..bc93bdb  master     -> origin/master
CONFLICT (delete/modify): alpha deleted in \
bc93bdb3e7cb46678166ee403c6dfc71ba32e06c and modified in HEAD. \
Version HEAD of alpha left in tree.
Automatic merge failed; fix conflicts and then commit the result.
$ cd ./git-files-destroy-modified/repos2
$ git status
alpha: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   alpha
#
no changes added to commit (use "git add" and/or "git commit -a")
$ ls
alpha    beta     epsilon/ gamma/
$ cat alpha
alpha, modified
$ git ls-files --unmerged
100644 4a58007052a65fbc2fc3f910f2855f45a4058e74 1       alpha
100644 bdbcaf36d1d757776eed566cac34641de540ee83 2       alpha
$ git cat-file blob 4a58007052a65fbc2fc3f910f2855f45a4058e74
alpha
$ git cat-file blob bdbcaf36d1d757776eed566cac34641de540ee83
alpha, modified
```

5.2.6 Files/Destroy/moved

Git detects a conflict only if the move of the file which the merge wants to delete has been committed. If the move is just staged in the index, git does a fast-forward merge which results in the renamed file staying present in the working copy.

When running *git-files-modify-destroyed.sh* (see 8.3) with the line committing alpha's rename in repos2 removed, git does a fast-forward merge and alpha's destruction gets converted into a no-op:

```
   9121dd2..4cf4ced  master     -> origin/master
Updating 9121dd2..4cf4ced
```

```
Fast forward
 alpha |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
 delete mode 100644 alpha
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   alpha.moved
#
$ ls
alpha.moved  beta         epsilon/     gamma/
```

However, Git correctly detects the tree conflict if the rename of of alpha is committed before doing the merge:

```
   055fba9..d6d176d  master     -> origin/master
CONFLICT (rename/delete): Renamed alpha->alpha.moved in HEAD \
and deleted in d6d176dee25c666f933e1e9512acbaf60a9e7b6b
Automatic merge failed; fix conflicts and then commit the result.
```

For some reason git does not indicate the alpha.moved as unmerged. I don't know whether this is a design decision or a bug.

```
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha.moved  beta         epsilon/     gamma/
$ git ls-files --unmerged
$ cat alpha.moved
alpha
```

5.2.7 Files/Copy1/no object

Git does not treat this as a conflict.

After running *git-files-copy1-no-object.sh* (8.19), we can see that git creates the copied file to the target branch:

```
   ec8fee8..6206b9c  master     -> origin/master
Finished one cherry-pick.
Created commit 7db5a4d: copied eta
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 eta.copied
```

```
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha       beta        epsilon/    eta.copied  gamma/
$ cat eta.copied
eta
```

5.2.8 Files/Copy1/destroyed

Git does not treat this as a conflict.

If the destruction of the file in the target branch is not committed prior to the merge, git will resurrect the file.

This can be seen by running *git-files-copy1-destroyed.sh* (8.5), with the line committing the destruction of alpha removed:

```
   7f9a194..94187fe  master     -> origin/master
Updating 7f9a194..94187fe
Fast forward
 alpha.copied |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 alpha.copied
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha         alpha.copied  beta          epsilon/      gamma/
$ cat alpha
alpha
$ cat alpha.copied
alpha
```

When the destruction of alpha is committed before the merge, alpha is not resurrected, but its copy is added to the target branch unconditionally:

```
   9aecedb..dfe0491  master     -> origin/master
Merge made by recursive.
 alpha.copied |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 alpha.copied
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha.copied  beta          epsilon/      gamma/
```

```
$ cat alpha.copied
alpha
```

5.2.9 Files/Copy2/created

Git treats this case as a text conflict.

Git will not allow us to do this merge into a locally modified working copy:

```
error: Entry 'alpha.copied' would be overwritten by merge. Cannot merge.
```

So *git-files-copy2-created.sh* (see 8.9), commits local changes to the target branch before merging.

When running the script, we can see that git reports a conflict and tries to textually merge the files which both try to occupy the same path, which results in a text conflict:

```
   1fe3d46..eb15879  master     -> origin/master
Auto-merged alpha.copied
CONFLICT (add/add): Merge conflict in alpha.copied
Automatic merge failed; fix conflicts and then commit the result.
$ git status
alpha.copied: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   alpha.copied
#
no changes added to commit (use "git add" and/or "git commit -a")
$ ls
alpha         alpha.copied  beta          epsilon/      gamma/
$ cat alpha.copied
<<<<<<< HEAD:alpha.copied
alpha.copied, but different
=======
alpha
>>>>>>> eb1587968a2af1158e85c7190ad1b0ae4813b5fc:alpha.copied
```

5.2.10 Files/Copy2/moved

Git detects the tree conflict.

Git will not allow us to do this merge into a locally modified working copy:

```
error: Entry 'alpha.copied' would be overwritten by merge. Cannot merge.
```

So *git-files-copy2-moved.sh* (see 8.13), commits local changes to the target branch before merging.

When running the script, we can see that git reports a tree conflict on alpha.copied:

```
   37112bf..42db38f  master     -> origin/master
CONFLICT (rename/add): Renamed beta->alpha.copied in HEAD. \
alpha.copied added in 42db38fe0667bfc85f56312021522f641bde9019
Added as alpha.copied~42db38fe0667bfc85f56312021522f641bde9019 instead
Automatic merge failed; fix conflicts and then commit the result.
$ git status
alpha.copied: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   alpha.copied
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       alpha.copied~42db38fe0667bfc85f56312021522f641bde9019
no changes added to commit (use "git add" and/or "git commit -a")
$ ls
alpha
alpha.copied
alpha.copied~42db38fe0667bfc85f56312021522f641bde9019
epsilon/
gamma/
$ cat alpha
alpha
$ cat alpha.copied
beta
$ cat alpha.copied\~42db38fe0667bfc85f56312021522f641bde9019
alpha
```

5.2.11 Files/Move1/no object

Git detects the tree conflict.

When running *git-files-move1-no-object.sh* (8.15), we can see that git reports a tree conflict. The renamed file is added to the target tree at the move target path.

```
   374ae52..92212fd  master     -> origin/master
CONFLICT (rename/delete): Renamed eta->eta.moved in 92212fd... \
moved eta and deleted in HEAD
Automatic cherry-pick failed.  After resolving the conflicts,
```

```
mark the corrected paths with 'git add <paths>' or 'git rm <paths>' \
and commit the result.
When commiting, use the option '-c 92212fd' to retain authorship and message.
$ cd ./git-files-move1-no-object/repos2
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   eta.moved
#
$ ls
alpha      beta       epsilon/   eta.moved  gamma/
$ cat eta.moved
eta
$ git ls-files --unmerged
$
```

5.2.12 Files/Move1/destroyed

Git detects a conflict only if the destruction of the file which is about to be moved by the merge has been committed. If the destruction is just staged in the index the moved file is created at the move target path.

When running *git-files-move1-destroyed.sh* (see 8.4) with the line committing alpha's destruction in repos2 removed, git does a fast-forward merge and alpha.moved gets created in the target branch:

```
   6d32918..56a8650  master     -> origin/master
Updating 6d32918..56a8650
Fast forward
 alpha => alpha.moved |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename alpha => alpha.moved (100%)
$ cd ./git-files-move1-destroyed/repos2
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha.moved  beta         epsilon/     gamma/
```

When the destruction of alpha is committed, git reports a tree conflict:

```
   24dc595..6b4e285  master     -> origin/master
CONFLICT (rename/delete): Renamed alpha->alpha.moved in \
6b4e2855a9b66dfde4350c8815c6e2050ffd6301 and deleted in HEAD
```

```
Automatic merge failed; fix conflicts and then commit the result.
$ cd ./git-files-move1-destroyed/repos2
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   alpha.moved
#
$ ls
alpha.moved  beta         epsilon/     gamma/
$ cat alpha.moved
alpha
$ git ls-files --unmerged
$
```

5.2.13 Files/Move1/moved

Git detects a conflict only of the conflicting rename made in the target branch is committed prior to the merge.

When running *git-files-move1-moved.sh* (see 8.7) with the line committing alpha's rename to repos2 removed, git does a fast-forward merge and alpha.moved1 gets created in the target branch (alpha.moved2 being the locally renamed file):

```
   13bb344..b51c6bf  master     -> origin/master
Updating 13bb344..b51c6bf
Fast forward
 alpha => alpha.moved1 |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename alpha => alpha.moved1 (100%)
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   alpha.moved2
#
$ ls
alpha.moved1  alpha.moved2  beta          epsilon/      gamma/
$ cat alpha.moved1
alpha
$ cat alpha.moved2
alpha
```

When the rename is committed before merging, the conflict is detected correctly:

```
   8377ea9..4c4c827  master     -> origin/master
CONFLICT (rename/rename): Rename "alpha"->"alpha.moved2" in branch "HEAD" \
rename "alpha"->"alpha.moved1" in "4c4c8279d6ef4831a7ee54d0837be7b233dff588"
Automatic merge failed; fix conflicts and then commit the result.
$ git status
alpha: needs merge
alpha.moved1: needs merge
alpha.moved2: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   alpha
#       unmerged:   alpha.moved1
#       unmerged:   alpha.moved2
#
no changes added to commit (use "git add" and/or "git commit -a")
$ ls
alpha.moved1  alpha.moved2  beta          epsilon/      gamma/
$ cat alpha.moved1
alpha
$ cat alpha.moved2
alpha
$ git ls-files --unmerged
100644 4a58007052a65fbc2fc3f910f2855f45a4058e74 1       alpha
100644 4a58007052a65fbc2fc3f910f2855f45a4058e74 3       alpha.moved1
100644 4a58007052a65fbc2fc3f910f2855f45a4058e74 2       alpha.moved2
```

5.2.14 Files/Move2/created

Git detects the tree conflict.

Git will not allow us to do this merge into a locally modified working copy:

```
error: Entry 'eta' would be overwritten by merge. Cannot merge.
```

So *git-files-move2-created.sh* (see 8.6), commits local changes to the target branch before merging.

When running the script, we can see that git detects the clashing filenames of the renamed file in the source branch (eta) and the newly created file in the target branch (also eta):

```
   9a9c26a..d5850ad  master     -> origin/master
CONFLICT (rename/add): Renamed alpha->eta in \
d5850ad1d2731bdd85aabbcc6651a1ca40397cad. eta added in HEAD
Added as eta~HEAD instead
```

```
Automatic merge failed; fix conflicts and then commit the result.
$ git status
eta: needs merge
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    alpha
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   eta
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       eta~HEAD
$ ls
beta      epsilon/  eta       eta~HEAD  gamma/
$ cat eta
eta, but different
$ cat eta\~HEAD
eta, but different
$ git ls-files --unmerged
100644 780f7e45ade62ad00ad77fd2ee28098454257993 2       eta
100644 4a58007052a65fbc2fc3f910f2855f45a4058e74 3       eta
$ git cat-file blob 780f7e45ade62ad00ad77fd2ee28098454257993
eta, but different
$ git cat-file blob 4a58007052a65fbc2fc3f910f2855f45a4058e74
alpha
```

5.2.15 Files/Move2/moved

Git detects the tree conflict.

Git will not allow us to do this merge into a locally modified working copy:

```
error: Entry 'eta' would be overwritten by merge. Cannot merge.
```

So *git-files-move2-moved.sh* (see 8.8), commits local changes to the target branch before merging.

When running the script, we can see that git detects the clashing filenames of the renamed files in both branches, both of which want to occupy the move target path:

```
CONFLICT (rename/add): Renamed alpha->eta in \
```

```
4e3a5d9de7925a02f91903976a87a9cceec18121. eta added in HEAD
Added as eta~HEAD instead
CONFLICT (rename/add): Renamed beta->eta in HEAD. \
eta added in 4e3a5d9de7925a02f91903976a87a9cceec18121
Added as eta~4e3a5d9de7925a02f91903976a87a9cceec18121 instead
Automatic merge failed; fix conflicts and then commit the result.
$ git status
eta: needs merge
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    alpha
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   eta
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       eta~4e3a5d9de7925a02f91903976a87a9cceec18121
#       eta~HEAD
$ ls
epsilon/
eta
eta~4e3a5d9de7925a02f91903976a87a9cceec18121
eta~HEAD
gamma/
$ cat eta
beta
$ cat eta\~4e3a5d9de7925a02f91903976a87a9cceec18121
alpha
$ cat eta\~HEAD
beta
$ git ls-files --unmerged
100644 65b2df87f7df3aeedef04be96703e55ac19c2cfb 2       eta
100644 4a58007052a65fbc2fc3f910f2855f45a4058e74 3       eta
$ git cat-file blob 65b2df87f7df3aeedef04be96703e55ac19c2cfb
beta
$ git cat-file blob 4a58007052a65fbc2fc3f910f2855f45a4058e74
alpha
```

5.2.16 Directories/Create/created

Git does not treat this scenario as a conflict.

After running *git-directories-create-created.sh* (see 8.28), we can see that git merged the content of the two independently added directories:

```
   003ceea..6470932  master     -> origin/master
Updating 003ceea..6470932
Fast forward
 eta/theta |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 eta/theta
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   eta/iota
#
$ ls
alpha    beta     epsilon/ eta/     gamma/
$ ls eta
iota   theta
```

5.2.17 Directories/Create/moved

Git does not treat this scenario as a conflict.

When running *git-directories-create-moved.sh* (see 8.2) with the line committing the rename made on the target branch removed, git will resurrect the renamed directory at it the move source path:

```
   3e63241..ac810ef  master     -> origin/master
Updating 3e63241..ac810ef
Fast forward
 eta/theta |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 eta/theta
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   eta/delta
#
$ ls
alpha    beta     epsilon/ eta/     gamma/
$ ls gamma/
delta
```

So the script does a commit before doing the merge. The merge result is that the contents of the newly created and the renamed directory are merged:

```
   e7097c7..a1f208a  master     -> origin/master
Merge made by recursive.
 eta/theta |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 eta/theta
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha    beta     epsilon/ eta/
$ ls eta
delta  theta
```

5.2.18 Directories/Destroy/modified

Git does not treat this scenario as a conflict.

After running *git-directories-destroy-modified.sh* (see 8.27), we can see that git deleted overlapping content of the directory gamma which was destroyed in the source branch and the modified directory gamma on the target branch:

```
   47ee556..f3ad6ee  master     -> origin/master
Updating 47ee556..f3ad6ee
Fast forward
 gamma/delta |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
 delete mode 100644 gamma/delta
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   gamma/iota
#
$ ls
alpha    beta     epsilon/ gamma/
$ ls gamma
iota
```

5.2.19 Directories/Destroy/moved

Git detects a conflict only of the conflicting rename made in the target branch is committed prior to the merge.

When running *git-directories-destroy-moved.sh* (see 8.22) with the line committing gamma's rename to repos2 removed, git does a fast-forward merge and the renamed directory is retained:

```
   8e2ac79..786cd93  master     -> origin/master
Updating 8e2ac79..786cd93
Fast forward
 gamma/delta |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
 delete mode 100644 gamma/delta
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   gamma.moved/delta
#
$ ls
alpha        beta         epsilon/     gamma.moved/
$ ls gamma.moved
delta
```

When committing the rename prior to running the merge, git warns about a tree conflict, but the merge result is the same. Note that git does not treat the gamma directory as the victim of the tree conflict it detects. Rather, a file inside the gamma directory is considered the victim. This is because git does not treat directories as first-class objects.

```
   f7f27a3..919689e  master     -> origin/master
CONFLICT (rename/delete): Renamed gamma/delta->gamma.moved/delta in HEAD \
and deleted in 919689ebefb4e5986076a9fef5038e763da7c53b
Automatic merge failed; fix conflicts and then commit the result.
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha        beta         epsilon/     gamma.moved/
$ ls gamma.moved
delta
```

5.2.20 Directories/Copy1/no object

Git does not treat this case as a tree conflict.

After running *git-directories-copy1-no-object.sh* (see 8.29), we can see that git adds the copy target object as it exists on the source branch to the target branch:

```
   0e141bc..abd61e9  master     -> origin/master
Finished one cherry-pick.
Created commit 7ac0bc7: copied eta
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 eta.copied/theta
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha       beta        epsilon/    eta.copied/ gamma/
$ ls eta.copied
theta
```

5.2.21 Directories/Copy1/destroyed

Git does not treat this as a conflict.

If the destruction of the directory in the target branch is not committed prior to the merge, git will resurrect the directory.

This can be seen by running *git-directories-copy1-destroyed.sh* (8.14), with the line committing the destruction of gamma removed:

```
   c8a194d..e6ab850  master     -> origin/master
Updating c8a194d..e6ab850
Fast forward
 gamma.copied/delta |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 gamma.copied/delta
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha         beta          epsilon/      gamma/        gamma.copied/
$ ls gamma
delta
$ ls gamma.copied
delta
```

When the destruction of gamma is committed before the merge, gamma is not resurrected, but its copy is added to the target branch unconditionally:

```
   99f0500..468f1f2  master     -> origin/master
Merge made by recursive.
 gamma.copied/delta |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

```
 create mode 100644 gamma.copied/delta
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha         beta          epsilon/      gamma.copied/
$ ls gamma.copied
delta
```

Essentially, the behaviour of git in this case does not seem to differ depending on whether the objects involved are files or directories.

5.2.22 Directories/Copy2/created

Git does not treat this case as a conflict.

After running *git-directories-copy2-created.sh* (see 8.23), we can see that git merges the content of the copied directory gamma.copied on the source branch (its content being the file delta) with the content of the newly created directory gamma on the target branch (its content being the file theta):

```
   9f62235..7dafc2e  master     -> origin/master
Updating 9f62235..7dafc2e
Fast forward
 gamma.copied/delta |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 gamma.copied/delta
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   gamma.copied/theta
#
$ ls
alpha         beta          epsilon/      gamma/        gamma.copied/
$ ls gamma.copied
delta  theta
$
```

5.2.23 Directories/Copy2/moved

Git does not treat this case as a conflict.

After running *git-directories-copy2-moved.sh* (see 8.25), we can see that git merges the content of the copy target directory in the source branch with the content of the locally

moved directory in the target branch (both directories are named gamma.copied):

```
   732edd0..0a9318d  master     -> origin/master
Updating 732edd0..0a9318d
Fast forward
 gamma.copied/delta |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 gamma.copied/delta
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   gamma.copied/zeta
#
$ ls
alpha         beta          epsilon/      gamma/        gamma.copied/
$ ls gamma.copied
delta  zeta
$ ls gamma
delta
$ ls epsilon
zeta
```

5.2.24 Directories/Move1/no object

Git detects the tree conflict.

After running *git-directories-move1-no-object.sh* (see 8.26), we can see that git reported a conflict.

Note that git does not treat the eta directory as the victim of the tree conflict it detects. Rather, a file inside the eta directory is considered the victim. This is because git does not treat directories as first-class objects.

```
   1d97a69..7ef1ae0  master     -> origin/master
CONFLICT (rename/delete): Renamed eta/theta->eta.moved/theta in 7ef1ae0... \
moved eta and deleted in HEAD
Automatic cherry-pick failed.  After resolving the conflicts,
mark the corrected paths with 'git add <paths>' or 'git rm <paths>' \
and commit the result.
When commiting, use the option '-c 7ef1ae0' to retain authorship and message.
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
```

```
#       new file:   eta.moved/theta
#
$ ls
alpha      beta       epsilon/   eta.moved/ gamma/
$ ls eta.moved
theta
$ git ls-files --unmerged
$
```

5.2.25 Directories/Move1/destroyed

Git detects this conflict only if the rename made on the target branch is committed prior to running the merge.

When running *git-directories-move1-destroyed.sh* (see 8.1) with the line committing the rename of gamma on the target branch removed, git does a fast-forward merge and the rename directory (gamma.moved) is created in the target branch unconditionally:

```
   6aa0727..5dd88a0  master     -> origin/master
Updating 6aa0727..5dd88a0
Fast forward
 {gamma => gamma.moved}/delta |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename {gamma => gamma.moved}/delta (100%)
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha        beta         epsilon/     gamma.moved/
$ ls gamma.moved
delta
```

If the removal is committed, however, git reports a tree conflict:

```
   a4a6581..e3e5a92  master     -> origin/master
CONFLICT (rename/delete): Renamed gamma/delta->gamma.moved/delta in \
e3e5a92f96b94a98d8d990001c44551df77e85ee and deleted in HEAD
Automatic merge failed; fix conflicts and then commit the result.
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   gamma.moved/delta
#
$ ls
alpha        beta         epsilon/     gamma.moved/
```

```
$ ls gamma.moved
delta
$ git ls-files --unmerged
$
```

Note that git does not treat the gamma directory as the victim of the tree conflict it detects. Rather, a file inside the gamma directory is considered the victim. This is because git does not treat directories as first-class objects.

5.2.26 Directories/Move1/moved

Git detects this conflict only if the rename done on the target branch is committed prior to running the merge.

When running *git-directories-move1-moved.sh* (see 8.20) with the line committing the rename of gamma on the target branch removed, git does a fast-forward merge and the renamed directory from the source branch (gamma.moved1) is created in the target branch unconditionally:

```
   87b52bf..7a40265  master     -> origin/master
Updating 87b52bf..7a40265
Fast forward
 {gamma => gamma.moved1}/delta |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename {gamma => gamma.moved1}/delta (100%)
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   gamma.moved2/delta
#
$ ls
alpha         beta          epsilon/      gamma.moved1/ gamma.moved2/
$ ls gamma.moved1
delta
$ ls gamma.moved2
delta
```

However, if the rename on the target branch is committed before the merge is run, git reports a tree conflict:

```
   b02ac60..b679eb5  master     -> origin/master
CONFLICT (rename/rename): Rename "gamma/delta"->"gamma.moved2/delta" \
in branch "HEAD" rename "gamma/delta"->"gamma.moved1/delta" in \
```

```
"b679eb521d5ff90d5b865c4cd01fcbfa6bccd4e4"
Automatic merge failed; fix conflicts and then commit the result.
$ git status
gamma.moved1/delta: needs merge
gamma.moved2/delta: needs merge
gamma/delta: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   gamma.moved1/delta
#       unmerged:   gamma.moved2/delta
#       unmerged:   gamma/delta
#
no changes added to commit (use "git add" and/or "git commit -a")
$ ls
alpha         beta          epsilon/      gamma.moved1/ gamma.moved2/
$ ls gamma.moved1
delta
$ ls gamma.moved2
delta
$ git ls-files --unmerged
100644 ab135eefea6f73b921c7fec469b5f0e9db86b910 3       gamma.moved1/delta
100644 ab135eefea6f73b921c7fec469b5f0e9db86b910 2       gamma.moved2/delta
100644 ab135eefea6f73b921c7fec469b5f0e9db86b910 1       gamma/delta
$ git cat-file blob ab135eefea6f73b921c7fec469b5f0e9db86b910
delta
```

Note that git does not treat the gamma directory as the victim of the tree conflict it detects. Rather, a file inside the gamma directory is considered the victim. This is because git does not treat directories as first-class objects.

5.2.27 Directories/Move2/created

Git does not treat this case as a conflict.

When *git-directories-move2-created.sh* (see 8.16) is run with the line committing the creation of the directory gamma.moved in the target branch removed, git merges its content with the content of the directory which was renamed to the same path on the source branch:

```
   6d81b24..2612b3a  master     -> origin/master
Updating 6d81b24..2612b3a
Fast forward
 {gamma => gamma.moved}/delta |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename {gamma => gamma.moved}/delta (100%)
```

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   gamma.moved/eta
#
$ ls
alpha        beta         epsilon/     gamma.moved/
$ ls gamma.moved
delta  eta
```

When committing the creation of gamma.moved on the source branch prior to running the merge, the merge result is the same, except that git also leaves behind the parent directory of the move source path, which is left empty:

```
   3af8d2b..24a789c  master     -> origin/master
Merge made by recursive.
 {gamma => gamma.moved}/delta |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename {gamma => gamma.moved}/delta (100%)
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha        beta         epsilon/     gamma/       gamma.moved/
$ ls gamma
$ ls gamma.moved
delta  eta
```

5.2.28 Directories/Move2/moved

Git does not treat this case as a conflict.

When *git-directories-move2-moved.sh* (see 8.21) is run with the line committing the renamed directory gamma.moved in the target branch removed, git merges its content with the content of the directory which was renamed to the same path on the source branch:

```
   158f1bc..ed14234  master     -> origin/master
Updating 158f1bc..ed14234
Fast forward
 {gamma => gamma.moved}/delta |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename {gamma => gamma.moved}/delta (100%)
$ git status
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   gamma.moved/zeta
#
$ ls
alpha        beta         epsilon/     gamma.moved/
$ ls gamma.moved
delta  zeta
```

When committing the rename of gamma.moved on the source branch prior to running the merge, the merge result is the same, except that git also leaves behind the parent directory of the move source path, which is left empty:

```
   66b8cab..854980a  master     -> origin/master
Merge made by recursive.
 {gamma => gamma.moved}/delta |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename {gamma => gamma.moved}/delta (100%)
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
alpha        beta         gamma/       gamma.moved/
$ ls gamma
$ ls gamma.moved
delta  zeta
```

5.3 Detailed results for Subversion

5.3.1 Files/Modify/no object

Subversion does not detect this tree conflict. No conflict is signalled.

After running *svn-files-modify-no-object.sh* (see 8.34), we can see that the modification is not applied at all, because the file which is found to be non-existent is simply skipped during the merge:

```
Skipped missing target: 'svn-files-modify-no-object/branch/eta'
```

5.3.2 Files/Modify/destroyed

Subversion does not detect this tree conflict. No conflict is signalled.

After running *svn-files-modify-destroyed.sh* (see 8.50), we can see that the modification is not applied at all, because the deleted file is simply skipped during the merge. Subversion considers it missing:

```
Skipped missing target: 'svn-files-modify-no-object/branch/alpha'
```

Committing the removal of alpha before doing the merge does not affect the merge result.

5.3.3 Files/Create/created

Subversion treats this case as a text conflict.

When running *svn-files-create-created.sh* (see 8.37), Subversion reports a text conflict during the merge:

```
Conflict discovered in 'svn-files-create-created/branch/eta'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options: p
--- Merging r2 through r3 into 'svn-files-create-created/branch':
C    svn-files-create-created/branch/eta
$ cat svn-files-create-created/branch/eta
<<<<<<< .working
eta, but different
=======
eta
>>>>>>> .merge-right.r3
```

Committing the creation of eta before doing the merge does not affect the merge result.

5.3.4 Files/Create/moved

Subversion treats this case as a text conflict.

When running *svn-files-create-moved.sh* (see 8.43), Subversion reports a text conflict during the merge:

```
Conflict discovered in 'svn-files-create-moved/branch/eta'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options: p
--- Merging r2 through r3 into 'svn-files-create-moved/branch/eta':
C    svn-files-create-moved/branch/eta
$ cat svn-files-create-moved/branch/eta
<<<<<<< .working
alpha
```

```
=======
eta
>>>>>>> .merge-right.r3
```

Committing the rename of alpha to eta before doing the merge does not affect the merge result.

5.3.5 Files/Destroy/modified

Subversion does not detect this tree conflict. No conflict is signalled.

When running *svn-files-destroy-modified.sh* (see 8.36), with the line committing the modification of the file alpha in the target branch, we can see that the destruction is not carried out, because the locally modified file is simply skipped during the merge:

```
Skipped 'svn-files-destroyed-modified/branch/alpha'
$ cat svn-files-destroyed-modified/branch/alpha
alpha, but modified
```

When committing the modification prior to doing the merge, the merge destroys the file alpha in the target branch unconditionally:

```
--- Merging r2 through r4 into 'svn-files-destroyed-modified/branch':
D    svn-files-destroyed-modified/branch/alpha
$ ls svn-files-destroyed-modified/branch
beta     epsilon/ gamma/
```

In neither case is a conflict being raised.

5.3.6 Files/Destroy/moved

Subversion does not detect this tree conflict. No conflict is signalled.

When running *svn-files-destroy-modified.sh* (see 8.36), Subversion skips the destruction of the renamed file in the target branch, because it considers the file missing:

```
Skipped missing target: 'svn-files-destroyed-moved/branch/alpha'
```

5.3.7 Files/Copy1/no object

Subversion does not detect this tree conflict. No conflict is signalled.

After running *svn-files-copy1-no-object.sh* (see 8.38), we can see that Subversion simply creates in the target branch the copied file as it exists in the source branch:

```
--- Merging r4 into 'svn-files-copy1-no-object/branch':
A    svn-files-copy1-no-object/branch/eta.copied
$ cat svn-files-copy1-no-object/branch/eta.copied
eta
```

This is done regardless of whether the file of which the copy was made does actually exist in the target branch.

5.3.8 Files/Copy1/destroyed

Subversion does not detect this tree conflict. No conflict is signalled.

After running *svn-files-copy1-destroyed.sh* (see 8.52), we can see that Subversion simply creates in the target branch the copied file as it exists in the source branch:

```
--- Merging r2 through r3 into 'svn-files-copy1-destroyed/branch':
A    svn-files-copy1-destroyed/branch/alpha.copied
$ cat svn-files-copy1-destroyed/branch/alpha.copied
alpha
```

This is done regardless of whether the file of which the copy was made does actually exist in the target branch.

5.3.9 Files/Copy2/created

Subversion treats this case as a text conflict.

When running *svn-files-copy2-created.sh* (see 8.56), Subversion reports a text conflict during the merge:

```
--- Merging r2 through r3 into 'svn-files-copy2-created/branch':
C    svn-files-copy2-created/branch/alpha.copied
$ cat svn-files-copy2-created/branch/alpha.copied
<<<<<<< .working
alpha.copied, different
=======
alpha
>>>>>>> .merge-right.r3
```

5.3.10 Files/Copy2/moved

Subversion does not detect a tree conflict. In fact, it fails to carry out the merge entirely. It just prints an error message.

After running *svn-files-copy2-moved.sh* (see 8.32), we can see that the merge command fails with the error:

```
svn: Working copy path 'alpha.copied' does not exist in repository
```

There seems to be no way to carry out the merge.

5.3.11 Files/Move1/no object

Subversion does not detect this tree conflict. No conflict is signalled.

After running *svn-files-move1-no-object.sh* (see 8.31), we can see that Subversion creates in the target branch the copied file as it exists in the source branch.

Also, Subversion's failed attempt to delete in the target branch the file at the move source path is made apparent by a "Skipped missing target" message.

```
--- Merging r4 into 'svn-files-move1-no-object/branch':
A    svn-files-move1-no-object/branch/eta.moved
Skipped missing target: 'svn-files-move1-no-object/branch/eta'
$ ls svn-files-move1-no-object/branch/
alpha      beta       epsilon/   eta.moved  gamma/
$ cat svn-files-move1-no-object/branch/eta.moved
eta
```

5.3.12 Files/Move1/destroyed

Subversion does not detect this tree conflict. No conflict is signalled.

After running *svn-files-move1-destroyed.sh* (see 8.51), we can see that Subversion creates in the target branch the moved file (alpha.moved) as it exists in the source branch.

Also, Subversion's failed attempt to delete in the target branch the file at the move source path is made apparent by a "Skipped missing target" message.

```
--- Merging r2 through r3 into 'svn-files-move1-destroyed/branch':
A    svn-files-move1-destroyed/branch/alpha.moved
Skipped missing target: 'svn-files-move1-destroyed/branch/alpha'
$ ls svn-files-move1-destroyed/branch
alpha.moved  beta
$ cat svn-files-move1-destroyed/branch/alpha.moved
alpha
```

5.3.13 Files/Move1/moved

Subversion does not detect the tree conflict.

After running *svn-files-move1-moved.sh* (see 8.54), we can see that Subversion creates the file at the target path of the rename which happened in the source branch. The result of the merge is that two identical files are created at two different paths in the target tree. Effectively, both the rename made on the source branch and the rename made on the target branch are carried out independently and are not considered to be conflicting.

Also, Subversion's failed attempt to delete in the target branch the file at the move source path is made apparent by a "Skipped missing target" message.

```
--- Merging r2 through r3 into 'svn-files-move1-moved/branch':
A    svn-files-move1-moved/branch/alpha.moved1
Skipped missing target: 'svn-files-move1-moved/branch/alpha'
$ ls svn-files-move1-moved/branch
alpha.moved1  alpha.moved2
$ cat svn-files-move1-moved/branch/alpha.moved1
alpha
$ cat svn-files-move1-moved/branch/alpha.moved2
alpha
```

5.3.14 Files/Move2/created

Subversion treats this case as a text conflict.

When running *svn-files-move2-created.sh* (see 8.53), Subversion reports a text conflict during the merge.

The output also shows that Subversion deletes from the target branch the file at the move source path.

```
Conflict discovered in 'svn-files-move2-created/branch/alpha.moved'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options: p
--- Merging r2 through r3 into 'svn-files-move2-created/branch':
C    svn-files-move2-created/branch/alpha.moved
D    svn-files-move2-created/branch/alpha
$ ls svn-files-move2-created/branch
alpha.moved                 beta
alpha.moved.merge-left.r0   epsilon/
alpha.moved.merge-right.r3  gamma/
alpha.moved.working
$ cat svn-files-move2-created/branch/alpha.moved
<<<<<<< .working
```

```
alpha.moved, different
=======
alpha
>>>>>>> .merge-right.r3
```

5.3.15 Files/Move2/moved

Subversion does not detect a tree conflict. In fact, it fails to carry out the merge entirely. It just prints an error message.

After running *svn-files-move2-moved.sh* (see 8.55), we can see that the merge command fails with the error:

```
svn: Working copy path 'alpha.moved' does not exist in repository
```

There seems to be no way to carry out the merge.

5.3.16 Directories/Create/created

Subversion does not detect the tree conflict.

When running *svn-directories-create-created.sh* (see 8.47), Subversion merges the content of the independently created directories.

Subversion prints a "skipped" message for the directory eta, probably indicating that Subversion tried to create the directory in the target branch during the merge, but failed because the directory was already present.

```
Skipped 'svn-directories-create-created/branch/eta'
--- Merging r2 through r3 into 'svn-directories-create-created/branch':
A    svn-directories-create-created/branch/eta/iota
$ ls svn-directories-create-created/branch/eta/
iota   theta
```

5.3.17 Directories/Create/moved

Subversion does not detect the tree conflict.

When running *svn-directories-create-moved.sh* (see 8.49), Subversion merges the content of the independently renamed directories.

Subversion prints a "skipped" message for the directory eta, probably indicating that Subversion tried to create the directory in the target branch during the merge, but failed because the directory was already present.

```
Skipped 'svn-directories-create-moved/branch/eta'
--- Merging r2 through r3 into 'svn-directories-create-moved/branch/eta':
A    svn-directories-create-moved/branch/eta/iota
$ ls ./svn-directories-create-moved/branch/eta
delta  iota
```

5.3.18 Directories/Destroy/modified

Subversion does not detect the tree conflict.

The merge result depends on whether the addition of a file to the directory gamma on the target branch is committed prior to the merge.

In case no commit is made before running the merge, Subversion prints a "skipped" message for the directory gamma, and the locally added file iota is left intact:

```
A         svn-directories-destroy-modified/branch/gamma/iota
Skipped 'svn-directories-destroy-modified/branch/gamma'
$ ls svn-directories-destroy-modified/branch/gamma
delta  iota
$ svn status svn-directories-destroy-modified/branch/
 M     svn-directories-destroy-modified/branch
 M     svn-directories-destroy-modified/branch/gamma
A      svn-directories-destroy-modified/branch/gamma/iota
```

When a commit is made prior to the merge, the gamma directory is destroyed in the target branch unconditionally. The directory is left behind empty on disk, however:

```
--- Merging r2 through r4 into 'svn-directories-destroy-modified/branch':
D    svn-directories-destroy-modified/branch/gamma
$ ls svn-directories-destroy-modified/branch/gamma
$ ls svn-directories-destroy-modified/branch
alpha    beta     epsilon/ gamma/
$ svn status svn-directories-destroy-modified/branch/
 M     svn-directories-destroy-modified/branch
D      svn-directories-destroy-modified/branch/gamma
D      svn-directories-destroy-modified/branch/gamma/delta
D      svn-directories-destroy-modified/branch/gamma/iota
```

5.3.19 Directories/Destroy/moved

Subversion does not detect the tree conflict.

The merge result depends on whether the rename of the directory gamma on the target branch is committed prior to the merge.

If no commit is made, Subversion leaves the directory at the rename target path (gamma.moved) alone, and the deletion of the directory at the rename source path is transformed into a no-op:

```
--- Merging r2 through r3 into 'svn-directories-destroy-moved/branch':
D    svn-directories-destroy-moved/branch/gamma
$ ls svn-directories-destroy-moved/branch/
alpha        beta         epsilon/     gamma/       gamma.moved/
$ ls svn-directories-destroy-moved/branch/gamma
$ ls svn-directories-destroy-moved/branch/gamma.moved
delta
$ svn status svn-directories-destroy-moved/branch/
 M     svn-directories-destroy-moved/branch
D      svn-directories-destroy-moved/branch/gamma
D      svn-directories-destroy-moved/branch/gamma/delta
A  +   svn-directories-destroy-moved/branch/gamma.moved
```

When the rename is committed before the merge is run, Subversion prints a "skipped" message for the directory gamma during the merge. This indicates that it tried to run a delete operation on the directory which failed because the directory was absent:

```
Skipped missing target: 'svn-directories-destroy-moved/branch/gamma'
$ ls svn-directories-destroy-moved/branch
alpha        beta         epsilon/     gamma.moved/
$ ls svn-directories-destroy-moved/branch/gamma.moved
delta
$ svn status svn-directories-destroy-moved/branch/
 M     svn-directories-destroy-moved/branch
 M     svn-directories-destroy-moved/branch/gamma.moved
```

The modifications shown in the output of the status command indicated property modifications (most probably mergeinfo), and not content modifications.

5.3.20 Directories/Copy1/no object

Subversion does not treat this case as a conflict.

After running *svn-directories-copy1-no-object.sh* (see 8.48), we can see that Subversion creates the directory at the copy target path in the target branch unconditionally:

```
--- Merging r4 into 'svn-directories-copy1-no-object/branch':
A    svn-directories-copy1-no-object/branch/eta.copied
$ svn status svn-directories-copy1-no-object/branch
 M     svn-directories-copy1-no-object/branch
A  +   svn-directories-copy1-no-object/branch/eta.copied
```

```
$ ls svn-directories-copy1-no-object/branch/
alpha       beta        epsilon/    eta.copied/ gamma/
```

5.3.21 Directories/Copy1/destroyed

Subversion does not detect the tree conflict.

After running *svn-directories-copy1-destroyed.sh* (see 8.33), we can see that Subversion creates the directory gamma.copied in the target branch, and leaves the locally destroyed directory gamma alone:

```
--- Merging r2 through r3 into 'svn-directories-copy1-destroyed/branch':
A    svn-directories-copy1-destroyed/branch/gamma.copied
A    svn-directories-copy1-destroyed/branch/gamma.copied/delta
$ svn status svn-directories-copy1-destroyed/branch/
 M     svn-directories-copy1-destroyed/branch
D      svn-directories-copy1-destroyed/branch/gamma
D      svn-directories-copy1-destroyed/branch/gamma/delta
A  +   svn-directories-copy1-destroyed/branch/gamma.copied
A  +   svn-directories-copy1-destroyed/branch/gamma.copied/delta
```

5.3.22 Directories/Copy2/created

Subversion does not detect the tree conflict.

After running *svn-directories-copy2-created.sh* (see 8.42), we can see that Subversion unconditionally merges the independently created directories in the target branch:

```
--- Merging r2 through r3 into 'svn-directories-copy2-created/branch':
A    svn-directories-copy2-created/branch/gamma.copied/delta
$ svn status svn-directories-copy2-created/branch
 M     svn-directories-copy2-created/branch
A      svn-directories-copy2-created/branch/gamma.copied
A  +   svn-directories-copy2-created/branch/gamma.copied/delta
A      svn-directories-copy2-created/branch/gamma.copied/iota
```

5.3.23 Directories/Copy2/moved

Subversion does not detect a tree conflict. In fact, it fails to carry out the merge entirely. It just prints an error message.

After running *svn-directories-copy2-created.sh* (see 8.42), we can see that the merge command fails with the error:

```
svn: Working copy path 'gamma.copied' does not exist in repository
```

There seems to be no way to carry out the merge.

5.3.24 Directories/Move1/no object

Subversion does not detect the tree conflict.

After running *svn-directories-move1-no-object.sh* (see 8.45), we can see that the directory eta.copied is added to the target branch unconditionally as it exists in the source branch:

```
--- Merging r4 into 'svn-directories-move1-no-object/branch':
A    svn-directories-move1-no-object/branch/eta.copied
$ svn status svn-directories-move1-no-object/branch/
 M     svn-directories-move1-no-object/branch
A  +   svn-directories-move1-no-object/branch/eta.copied
```

5.3.25 Directories/Move1/destroyed

Subversion does not detect this tree conflict. No conflict is signalled.

After running *svn-directories-move1-destroyed.sh* (see 8.57), we can see that Subversion creates in the target branch the moved directory (gamma.moved) as it exists in the source branch.

```
--- Merging r2 through r3 into 'svn-directories-move1-destroyed/branch':
A    svn-directories-move1-destroyed/branch/gamma.moved
A    svn-directories-move1-destroyed/branch/gamma.moved/delta
D    svn-directories-move1-destroyed/branch/gamma
$ svn status svn-directories-move1-destroyed/branch/
 M     svn-directories-move1-destroyed/branch
D      svn-directories-move1-destroyed/branch/gamma
D      svn-directories-move1-destroyed/branch/gamma/delta
A  +   svn-directories-move1-destroyed/branch/gamma.moved
A  +   svn-directories-move1-destroyed/branch/gamma.moved/delta
```

5.3.26 Directories/Move1/moved

Subversion does not treat this case as a tree conflict.

After running *svn-directories-move1-moved.sh* (see 8.39), we can see that Subversion unconditionally creates the directory gamma.moved1 (the target path of the rename made in the source branch) in the target branch. The result of the merge is that two identical directories are created at two different paths in the target tree. Effectively, both the rename made on the source branch and the rename made on the target branch are carried out independently and are not considered to be conflicting.

```
--- Merging r2 through r3 into 'svn-directories-move1-moved/branch':
A    svn-directories-move1-moved/branch/gamma.moved1
A    svn-directories-move1-moved/branch/gamma.moved1/delta
D    svn-directories-move1-moved/branch/gamma
$ svn status svn-directories-move1-moved/branch
 M     svn-directories-move1-moved/branch
D      svn-directories-move1-moved/branch/gamma
D      svn-directories-move1-moved/branch/gamma/delta
A  +   svn-directories-move1-moved/branch/gamma.moved1
A  +   svn-directories-move1-moved/branch/gamma.moved1/delta
A  +   svn-directories-move1-moved/branch/gamma.moved2
$ ls svn-directories-move1-moved/branch/
alpha         epsilon/      gamma.moved1/
beta          gamma/        gamma.moved2/
$ ls svn-directories-move1-moved/branch/gamma.moved1
delta
$ ls svn-directories-move1-moved/branch/gamma.moved2
delta
```

5.3.27 Directories/Move2/created

Subversion does not treat this case as a tree conflict.

After running *svn-directories-move2-created.sh* (see 8.35), we can see that Subversion unconditionally merges the content of the independently renamed and created directories.

```
Skipped 'svn-directories-move2-created/branch/gamma.moved'
--- Merging r2 through r3 into 'svn-directories-move2-created/branch':
A    svn-directories-move2-created/branch/gamma.moved/delta
D    svn-directories-move2-created/branch/gamma
$ svn status svn-directories-move2-created/branch/
 M     svn-directories-move2-created/branch
D      svn-directories-move2-created/branch/gamma
D      svn-directories-move2-created/branch/gamma/delta
A      svn-directories-move2-created/branch/gamma.moved
A  +   svn-directories-move2-created/branch/gamma.moved/delta
A      svn-directories-move2-created/branch/gamma.moved/iota
```

5.3.28 Directories/Move2/moved

Subversion does not detect a tree conflict. In fact, it fails to carry out the merge entirely. It just prints an error message.

After running *svn-directories-move2-moved.sh* (see 8.40), we can see that the merge command fails with the error:

```
svn: Working copy path 'gamma.moved' does not exist in repository
```

There seems to be no way to carry out the merge.

6 Possible Improvements to the Model

In spite of the shortcomings described in this section, I am nevertheless happy with the final iteration of the model. I am convinced that it is suitable to illustrate a reasonably-sized chunk of the tree conflict problem space, in an implementation-independent manner. That is all I needed it for, so it served its purpose well.

6.1 Renamed and deleted parent directories

The model as presented in section 2 currently cannot express merge scenarios involving renamed or deleted parent directories, a deficiency pointed out to me twice, once by Nico Schellingerhout and once by Julian Foad, independently.

When creating an object, which may happen as either a stand-alone operation or as part of a copy or move operation, the destination directory is assumed to exist in the target tree at the same path as it does in the source tree. This fails to account for deleted or renamed directories in the target tree.

One way of fixing this would be to split the create operation into two distinct steps – one step which would try to locate the parent directory of the new object by the corresponding parent directory's ID in the source tree, and another step which would ensure that the directory entry to be used was not already occupied by a different object (otherwise, two objects could meet in the tree under the same path name).

When Nico pointed out the problem, I did not deem it too serious, because it seemed to me that I already had a sufficient number of tree conflict cases which could be described.

When Julian pointed out the same problem again much later, I was finally convinced to give it a try and considered changing the model. But the deadline was too close to allow for the necessary changes to be made.

Breaking up the create operation into two distinct steps would have added another step to the copy and move operations. I did not have enough time to adjust all the scripts dealing with these cases. Also, new scripts would have had to be written to account for additional tree conflict cases involving missing or renamed parent directories. Results already obtained would have had to be adapted or extended. All this was not possible in the small bit of time I had left before deadline, which I had to use to make the finishing touches required to get the text in shape for submission.

6.2 Flags

The concept of "flags" is an evolutionary left-over from an early iteration of the model, which attached flags to paths to describe local changes made in the target line of history. In this iteration, all operations were locating objects purely by path instead of ID, and thus a concept of historical meta data about objects which could be located via paths was required.

A worthwhile improvement would probably be to get rid of the concept of flags, in favour of adding a "set of operations" to the "merge scenario". Such a set would consist of the operations which have been performed on an object in the target line of history, but not in the source line of history. Essentially, this set is what flags are supposed to represent. Using this set directly instead of an abstraction which represents it would probably make the model easier to explain and understand.

The idea of getting rid of the concept of flags entirely occurred to me only really close before deadline. I had no chance to really think it through yet, let alone adapt the model in time for submission.

7 Acknowledgments

7.1 People

My mentor Martin Gruhn and professor Lutz Prechelt have been extraordinarily kind and patient, and have granted me an unbelievable amount of creative freedom for my thesis. Martin also discussed with me the many different iterations of the tree conflict model, and even though he bluntly stated that version control systems were not his primary field of expertise, he managed to discover quite a few flaws in the model by asking insightful questions. He has also helped me a lot with structuring the text.

Julian Foad, Nico Schellingerhout, Stephen Butler, Neels Janosch Hofmeyr, Erik Hülsmann, Olaf Wagner, Michael Diers, and Olaf Kosel all provided valuable feedback on several draft versions of this text. Countless spelling errors and various conundrums readers would have had to helplessly try to understand were avoided because of their help. They especially helped me by spotting errors and inconsistencies in the model, and suggesting alternative approaches which I otherwise would not have thought of.

Karl Fogel honoured me by printing a draft version of this text on a printer at the softwarefreedom.org offices. He also pointed out how to convince LaTeX to render a blank line between paragraphs, saving anyone reading this text a good amount of eye strain. Karl is doing important non-profit work at questioncopyright.org[11]. Browsing that website reminded me to release this text under a free licence.

Special thanks go to Elego Software Solutions[12] for funding me while I was working

on my thesis. Their support is very much appreciated, and goes well beyond financial aspects. Their experience and expertise in anything related to version control have been a very valuable resource for many years.

7.2 Other interesting version control systems

I would be glad to see scripts which reproduce my tree conflict cases with other version control systems. A few interesting systems are mentioned below, but I am sure there are many more.

I considered all the tools listed, but could not examine them because of lack of time.

7.2.1 ClearCase

IBM Rational ClearCase[13] is a proprietary SCM system which has been personally recommended to me by a few of its users for its powerful merge support.

However, the system is rather complicated to set up, and I could not get access to an existing installation.

7.2.2 MolhadoRef

MolhadoRef[20] is an implementation of a "Refactoring-aware SCM system" for use with the Java programming language.

Interestingly, MolhadoRef shares some properties with the tree conflict model presented here. It is ID-based, aware of program semantics, and operation-based, in order to "automatically detect more merge conflicts than CVS."[20]

7.2.3 Bazaar

Bazaar[14] is a distributed version control system distributed under the GNU General public licence.

According to the project's website "Bazaar has perfect support for renaming files AND directories. This means developers can refactor without holding back because of fear of merging."

7.2.4 AccuRev

AccuRev[15] is a proprietary version control system which claims to have "support to track and merge changes involving file and directory moves and renames."[16]

7.2.5 Codeville

Codeville[17] is a distributed version control system, released under an open source licence.

According to the project's website, it implements "file and directory renaming which correctly handles all (and there are plenty) corner cases and conflicts."

References

[1] http://java.sun.com (accessed 28th of August 2008).

[2] http://www.python.org (accessed 28th of August 2008).

[3] http://subversion.tigris.org/ (accessed 28th of August 2008).

[4] http://www.selenic.com/mercurial/ (acccessed 28th of August 2008).

[5] http://www.selenic.com/mercurial/wiki/index.cgi/TransplantExtension (accessed September 1st 2008).

[6] http://git.org.cz/ (accessed 31st of August 2008).

[7] http://git.or.cz/gitwiki/ContentLimitations?action=recall&rev=10 (accessed 31st of August 2008).

[8] http://www.kernel.org/pub/software/scm/git/docs/user-manual.html#the-index (accessed 31st of August 2008).

[9] Subversion issue #3067, "subtrees that don't exist at the start or end of a merge range shouldn't break the merge", http://subversion.tigris.org/issues/show_bug.cgi?id=3067 (accessed September 1st 2008).

[10] Mercurial issue #1175, "KeyError for copy in commit", http://www.selenic.com/mercurial/bts/issue1175 (accessed 1st September 2008).

[11] "A Clearinghouse For New Ideas About Copyright", http://questioncopyright.org/ (accessed September 1st 2008).

[12] Elego Software Solutions GmbH, Berlin, Germany, http://www.elego.de/ (accessed September 1st 2008).

[13] http://www-01.ibm.com/software/awdtools/clearcase/ (accessed 28th of August 2008).

[14] http://bazaar-vcs.org/ (accessed 28th of August 2008).

[15] http://www.accurev.com/ (accessed 28th of August 2008).

[16] http://www.accurev.com/scm_comparisons.html (accessed 28th of August 2008).

[17] http://codeville.org/ (accessed 28th of August 2008).

[18] Brian Berliner. CVS II: Parallelizing software development. *USENIX Conference Proceedings*, pages 341–352, 1990. http://www.fnal.gov/docs/products/cvs/cvs-paper.ps (accessed 31st of August 2008).

[19] Valdis Berzins. Software Merge: Semantics of Combining Changes to Programs. *ACM Transactions on Programming Languages and Systems*, 16(6):1875–1903, November 1994.

[20] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering*, 34(3), May/June 2008.

[21] Julian Foad, Nico Schellingerhout, and Stefan Sperling. Analysis of the 30 test cases (for tree conflicts). Discussion on the Subversion development mailing list, Date: Sun, 20 Apr 2008 10:51:49 +0200, Message-ID: OF8D84FCC6.3C08D781-ONC1257431.00306971-C1257431.0030B07C@philips.com, http://subversion.tigris.org/servlets/ReadMsg?listName=dev&msgNo=137477 (accessed 16th of May 2008).

[22] Graydon Hoare et al. *Monotone, a distributed version control system.* no publisher. This edition documents version 0.40. Made available under the GNU GPL version 2.0 or greater. http://monotone.ca/monotone.pdf (accessed 21st of July 2008).

[23] Tancred Lindholm. A Three-way Merge for XML Documents. *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 1–10, 2004.

[24] Ernst Lippe and Norbert van Oosterom. Operation-Based Merging. *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, 17(5):78–87, 1992.

[25] Tom Mens. A state-of-the-Art survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[26] Bryan O'Sullivan. *Distributed revision control with Mercurial.* no publisher, December 2007. rev 4700dd38384c, full text available under the Open Publication License at http://hgbook.red-bean.com.

[27] Nico Schellingerhout, Stephen Butler, and Stefan Sperling. Tree Conflicts Use Cases and Desired Behaviours. Subversion-related tree conflict use case notes, http://svn.collab.net/viewvc/svn/trunk/notes/tree-conflicts/use-cases.txt?revision=28885 (accessed 28th of August 2008).

[28] Walter F. Tichy. Tools for Software Configuration Management. *Proc. Int'l Workshop Software Version and Configuration Control*, pages 1–20, 1988.

[29] Linus Torvalds. Re: impure renames / history tracking. Date: 2006-03-01 17:13:53, Message-ID: Pine.LNX.4.64.0603010859200.22647@g5.osdl.org, http://marc.info/?l=git&m=114123702826251 (accessed 1st September 2008).

8 Appendix A - Scripts

Licence

```
I wrote these scripts for my Bachelor thesis.
They are licensed as follows:

Copyright (c) 2008 Stefan Sperling <sperling@inf.fu-berlin.de>

Permission to use, copy, modify, and distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

8.1 scripts/git/git-directories-move1-destroyed.sh

```
#!/bin/sh
# $Id: git-directories-move1-destroyed.sh 178 2008-08-14 13:57:02Z stsp $

. ./git-init.sh

# Move a directory in repos
(cd $repos && git mv gamma gamma.moved)
(cd $repos && git commit -a -m "moved gamma")

# Remove the same directory in repos2.
(cd $repos2 && git rm -r gamma)
# We need to commit here, else git won't detect the conflict
(cd $repos2 && git commit -a -m "destroyed gamma")

# Pull the move into repos2 and merge
(cd $repos2 && git pull)
```

8.2 scripts/git/git-directories-create-moved.sh

```
#!/bin/sh
# $Id: git-directories-create-moved.sh 163 2008-08-12 14:31:30Z stsp $

. ./git-init.sh
```

```
# Create a directory in repos
mkdir $repos/eta
echo "theta" > $repos/eta/theta
(cd $repos && git add eta)
(cd $repos && git commit -m "added eta")

# Move a different directory in repos2 to the same path
(cd $repos2 && git mv gamma eta)
# We have to commit here, else git will resurrect gamma during the merge
(cd $repos2 && git commit -m "renamed gamma to eta")

# Pull the add into repos2 and merge
(cd $repos2 && git pull)
```

8.3 scripts/git/git-files-modify-destroyed.sh

```
#!/bin/sh
# $Id: git-files-modify-destroyed.sh 152 2008-08-11 15:55:36Z stsp $

. ./git-init.sh

# Modify a file in repos
echo "alpha, modified" > $repos/alpha
(cd $repos && git add alpha)
(cd $repos && git commit -a -m "changed alpha")

# Destroy the same file in repos2
(cd $repos2 && git rm alpha)
# We need to commit here, else git won't detect a conflict.
(cd $repos2 && git commit -a -m "remove alpha")

# Pull the modified file into repos2 and merge
(cd $repos2 && git pull)
```

8.4 scripts/git/git-files-move1-destroyed.sh

```
#!/bin/sh
# $Id: git-files-move1-destroyed.sh 178 2008-08-14 13:57:02Z stsp $

. ./git-init.sh

# Move a file in repos
(cd $repos && git mv alpha alpha.moved)
(cd $repos && git commit -a -m "moved alpha")

# Remove the same file in repos2.
(cd $repos2 && git rm alpha)
# We need to commit here, else git won't detect the conflict
(cd $repos2 && git commit -a -m "destroyed alpha")

# Pull the move into repos2 and merge
```

```
(cd $repos2 && git pull)
```

8.5 scripts/git/git-files-copy1-destroyed.sh

```
#!/bin/sh
# $Id: git-files-copy1-destroyed.sh 152 2008-08-11 15:55:36Z stsp $

. ./git-init.sh

# Copy a file in repos.
cp $repos/alpha $repos/alpha.copied
(cd $repos && git add alpha.copied)
(cd $repos && git commit -a -m "copied alpha")

# Destroy the same file in repos2
(cd $repos2 && git rm alpha)
# We need to commit here, else alpha will be resurrected by the merge
(cd $repos2 && git commit -a -m "destroyed alpha")

# Pull the copy into repos2 and merge
(cd $repos2 && git pull)
```

8.6 scripts/git/git-files-move2-created.sh

```
#!/bin/sh
# $Id: git-files-move2-created.sh 158 2008-08-11 19:19:37Z stsp $

. ./git-init.sh

# Move a file in repos
(cd $repos && git mv alpha eta)
(cd $repos && git commit -a -m "renamed alphad")

# Create a file at the move target path in repos2
echo "eta, but different" > $repos2/eta
(cd $repos2 && git add eta)
# We need to commit here, else git won't detect a conflict.
(cd $repos2 && git commit -a -m "renamed alpha, too")

# Pull the rename into repos2 and merge
(cd $repos2 && git pull)
```

8.7 scripts/git/git-files-move1-moved.sh

```
#!/bin/sh
# $Id: git-files-move1-moved.sh 157 2008-08-11 19:09:37Z stsp $

. ./git-init.sh
```

```
# Move a file in repos
(cd $repos && git mv alpha alpha.moved1)
(cd $repos && git commit -a -m "renamed alphad")

# Rename the same file to a different name in repos2
(cd $repos2 && git mv alpha alpha.moved2)
# We need to commit here, else git won't detect a conflict.
(cd $repos2 && git commit -a -m "renamed alpha, too")

# Pull the rename into repos2 and merge
(cd $repos2 && git pull)
```

8.8 scripts/git/git-files-move2-moved.sh

```
#!/bin/sh
# $Id: git-files-move2-moved.sh 181 2008-08-14 14:23:39Z stsp $

. ./git-init.sh

# Move a file in repos
(cd $repos && git mv alpha eta)
(cd $repos && git commit -a -m "renamed alpha")

# Move a file different file in repos2 to the move target path
(cd $repos2 && git mv beta eta)
# We need to commit here, else git won't detect a conflict.
(cd $repos2 && git commit -a -m "renamed beta to the same move target path")

# Pull the rename into repos2 and merge
(cd $repos2 && git pull)
```

8.9 scripts/git/git-files-copy2-created.sh

```
#!/bin/sh
# $Id: git-files-copy2-created.sh 153 2008-08-11 16:10:28Z stsp $

. ./git-init.sh

# Copy a file in repos.
cp $repos/alpha $repos/alpha.copied
(cd $repos && git add alpha.copied)
(cd $repos && git commit -a -m "copied alpha")

# Create a file at the copy target path in repos2
echo "alpha.copied, but different" > $repos2/alpha.copied
(cd $repos2 && git add alpha.copied)
# We need to commit here, else git won't merge
(cd $repos2 && git commit -a -m "created alpha.copied")

# Pull the copy into repos2 and merge
(cd $repos2 && git pull)
```

8.10 scripts/git/git-files-modify-no-object.sh

```
#!/bin/sh
# $Id: git-files-modify-no-object.sh 147 2008-08-11 14:13:15Z stsp $

. ./git-init.sh

# Add a file eta in repos
echo "eta" > $repos/eta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "added eta")

# Modify eta in repos
echo "eta, modified" > $repos/eta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "changed eta")

# Cherry-pick just the modification of eta into repos2
(cd $repos2 && git fetch)
rev=`(cd $repos2 && git log -1 --pretty=oneline origin | cut -d ' ' -f 1)`
(cd $repos2 && git cherry-pick $rev)
```

8.11 scripts/git/git-files-destroy-moved.sh

```
#!/bin/sh
# $Id: git-files-destroy-moved.sh 150 2008-08-11 15:17:38Z stsp $

. ./git-init.sh

# Destroy a file in repos
(cd $repos && git rm alpha)
(cd $repos && git commit -m "destroyed alpha")

# Move the same file in repos2
# Make the modification visible in the index
(cd $repos2 && git mv alpha alpha.moved)
# We need to commit here, else git won't detect the conflict
(cd $repos2 && git commit -m "changed alpha")

# Pull the destruction of alpha into repos2 and merge
(cd $repos2 && git pull)
```

8.12 scripts/git/git-init.sh

```
#!/bin/sh
# $Id: git-init.sh 113 2008-08-02 15:21:08Z stsp $

set -e

basename=`basename $0`
scratch_area="`echo $basename | sed -e s/\.sh$//`"
```

```
repos=$scratch_area/repos
repos2=$scratch_area/repos2

rm -rf $scratch_area

/bin/sh ../projtree.sh $repos

(cd $repos && git init && git add .)
(cd $repos && git commit -a -m "importing project tree")
git clone $repos $repos2
```

8.13 scripts/git/git-files-copy2-moved.sh

```
#!/bin/sh
# $Id: git-files-copy2-moved.sh 154 2008-08-11 16:33:31Z stsp $

. ./git-init.sh

# Copy a file in repos.
cp $repos/alpha $repos/alpha.copied
(cd $repos && git add alpha.copied)
(cd $repos && git commit -a -m "copied alpha")

# Move a file to the copy target path in repos2
(cd $repos2 && git mv beta alpha.copied)
# We need to commit here, else git won't merge
(cd $repos2 && git commit -a -m "created alpha.copied")

# Pull the copy into repos2 and merge
(cd $repos2 && git pull)
```

8.14 scripts/git/git-directories-copy1-destroyed.sh

```
#!/bin/sh
# $Id: git-directories-copy1-destroyed.sh 169 2008-08-14 11:41:20Z stsp $

. ./git-init.sh

# Copy a directory in repos.
cp -r $repos/gamma $repos/gamma.copied
(cd $repos && git add gamma.copied)
(cd $repos && git commit -a -m "copied gamma")

# Destroy the same directory in repos2
(cd $repos2 && git rm -r gamma)
# We need to commit here, else gamma will be resurrected by the merge
(cd $repos2 && git commit -a -m "destroyed gamma")

# Pull the copy into repos2 and merge
(cd $repos2 && git pull)
```

8.15 scripts/git/git-files-move1-no-object.sh

```
#!/bin/sh
# $Id: git-files-move1-no-object.sh 155 2008-08-11 17:19:50Z stsp $

. ./git-init.sh

# Add a file in repos
echo "eta" > $repos/eta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "added eta")

# Move the new file.
(cd $repos && git mv eta eta.moved)
(cd $repos && git commit -a -m "moved eta")

# Cherry-pick just the move into repos2
(cd $repos2 && git fetch)
rev=`(cd $repos2 && git log -1 --pretty=oneline origin | cut -d ' ' -f 1)`
(cd $repos2 && git cherry-pick $rev)
```

8.16 scripts/git/git-directories-move2-created.sh

```
#!/bin/sh
# $Id: git-directories-move2-created.sh 180 2008-08-14 14:16:09Z stsp $

. ./git-init.sh

# Move a directory in repos
(cd $repos && git mv gamma gamma.moved)
(cd $repos && git commit -a -m "renamed gammad")

# Create a directory at the move target path in repos2
mkdir $repos2/gamma.moved
echo "eta" > $repos2/gamma.moved/eta
(cd $repos2 && git add gamma.moved)
# Committing here affects the merge result, but git won't
# detect a conflict either way.
(cd $repos2 && git commit -a -m "renamed gamma, too")

# Pull the rename into repos2 and merge
(cd $repos2 && git pull)
```

8.17 scripts/git/git-files-destroy-modified.sh

```
#!/bin/sh
# $Id: git-files-destroy-modified.sh 126 2008-08-06 14:39:12Z stsp $

. ./git-init.sh

# Destroy a file in repos
```

```
(cd $repos && git rm alpha)
(cd $repos && git commit -m "destroyed alpha")

# Modify the same file in repos2
echo "alpha, modified" > $repos2/alpha
# Make the modification visible in the index
(cd $repos2 && git add alpha)
# We need to commit here, else git won't merge
(cd $repos2 && git commit -m "changed alpha")

# Pull the destruction of alpha into repos2 and merge
(cd $repos2 && git pull)
```

8.18 scripts/git/git-files-create-created.sh

```
#!/bin/sh
# $Id: git-files-create-created.sh 119 2008-08-05 11:30:08Z stsp $

. ./git-init.sh

# Add a file in repos
echo "eta" > $repos/eta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "added eta")

# Add a file with the same name but different content
# in repos2's working copy
echo "eta, but different" > $repos2/eta
(cd $repos2 && git add eta)
# We need to commit here, else git won't merge.
(cd $repos2 && git commit -a -m "added different eta")

# Pull the add into repos2 and merge
(cd $repos2 && git pull)
```

8.19 scripts/git/git-files-copy1-no-object.sh

```
#!/bin/sh
# $Id: git-files-copy1-no-object.sh 175 2008-08-14 12:38:46Z stsp $

. ./git-init.sh

# Add a file in repos
echo "eta" > $repos/eta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "added eta")

# Create a copy of the new file.
cp $repos/eta $repos/eta.copied
(cd $repos && git add eta.copied)
(cd $repos && git commit -a -m "copied eta")
```

```
# Cherry-pick just the copy into repos2
(cd $repos2 && git fetch)
rev=‘(cd $repos2 && git log -1 --pretty=oneline origin | cut -d ’ ’ -f 1)‘
(cd $repos2 && git cherry-pick $rev)
```

8.20 scripts/git/git-directories-move1-moved.sh

```
#!/bin/sh
# $Id: git-directories-move1-moved.sh 179 2008-08-14 14:06:22Z stsp $

. ./git-init.sh

# Move a directory in repos
(cd $repos && git mv gamma gamma.moved1)
(cd $repos && git commit -a -m "renamed gammad")

# Rename the same directory to a different name in repos2
(cd $repos2 && git mv gamma gamma.moved2)
# We need to commit here, else git won’t detect a conflict.
(cd $repos2 && git commit -a -m "renamed gamma, too")

# Pull the rename into repos2 and merge
(cd $repos2 && git pull)
```

8.21 scripts/git/git-directories-move2-moved.sh

```
#!/bin/sh
# $Id: git-directories-move2-moved.sh 181 2008-08-14 14:23:39Z stsp $

. ./git-init.sh

# Move a directory in repos
(cd $repos && git mv gamma gamma.moved)
(cd $repos && git commit -a -m "renamed gamma")

# Move a directory different directory in repos2 to the move target path
(cd $repos2 && git mv epsilon gamma.moved)
# Committing here affects the merge result, but git won’t
# detect a conflict either way.
(cd $repos2 && git commit -a -m "renamed epsilon to the same move target path")

# Pull the rename into repos2 and merge
(cd $repos2 && git pull)
```

8.22 scripts/git/git-directories-destroy-moved.sh

```
#!/bin/sh
# $Id: git-directories-destroy-moved.sh 165 2008-08-12 16:26:16Z stsp $
```

```
. ./git-init.sh

# Destroy a directory in repos
(cd $repos && git rm -r gamma)
(cd $repos && git commit -m "destroyed gamma")

# Rename the same directory in repos2
(cd $repos2 && git mv gamma gamma.moved)
# We need to commit here, else git does not detect the conflict.
(cd $repos2 && git commit -m "renamed gamma")

# Pull the destruction of alpha into repos2 and merge
(cd $repos2 && git pull)
```

8.23 scripts/git/git-directories-copy2-created.sh

```
#!/bin/sh
# $Id: git-directories-copy2-created.sh 171 2008-08-14 11:51:03Z stsp $

. ./git-init.sh

# Copy a directory in repos.
cp -r $repos/gamma $repos/gamma.copied
(cd $repos && git add gamma.copied)
(cd $repos && git commit -a -m "copied gamma")

# Create a directory at the copy target path in repos2
mkdir $repos2/gamma.copied
echo "theta" > $repos2/gamma.copied/theta
(cd $repos2 && git add gamma.copied)
# Committing here does not affect the merge result.

# Pull the copy into repos2 and merge
(cd $repos2 && git pull)
```

8.24 scripts/git/git-files-create-moved.sh

```
#!/bin/sh
# $Id: git-files-create-moved.sh 148 2008-08-11 14:58:24Z stsp $

. ./git-init.sh

# Add a file in repos
echo "eta" > $repos/eta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "added eta")

# Move a different file with the same path in repos2
(cd $repos2 && git mv alpha eta)
# We need to commit here, else git won't merge.
(cd $repos2 && git commit -a -m "added different eta")
```

```
# Pull the add into repos2 and merge
(cd $repos2 && git pull)
```

8.25 scripts/git/git-directories-copy2-moved.sh

```
#!/bin/sh
# $Id: git-directories-copy2-moved.sh 172 2008-08-14 12:26:53Z stsp $

. ./git-init.sh

# Copy a directory in repos.
cp -r $repos/gamma $repos/gamma.copied
(cd $repos && git add gamma.copied)
(cd $repos && git commit -a -m "copied gamma")

# Move a directory to the copy target path in repos2
(cd $repos2 && git mv epsilon gamma.copied)
# Committing here does not affect the merge result.

# Pull the copy into repos2 and merge
(cd $repos2 && git pull)
```

8.26 scripts/git/git-directories-move1-no-object.sh

```
#!/bin/sh
# $Id$

. ./git-init.sh

# Create a directory in repos
mkdir $repos/eta
echo "theta" > $repos/eta/theta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "added eta")

# Move the new directory.
(cd $repos && git mv eta eta.moved)
(cd $repos && git commit -a -m "moved eta")

# Cherry-pick just the move into repos2
(cd $repos2 && git fetch)
rev=‘(cd $repos2 && git log -1 --pretty=oneline origin | cut -d ’ ’ -f 1)‘
(cd $repos2 && git cherry-pick $rev)
```

8.27 scripts/git/git-directories-destroy-modified.sh

```
#!/bin/sh
# $Id: git-directories-destroy-modified.sh 164 2008-08-12 14:38:20Z stsp $
```

```
. ./git-init.sh

# Destroy a directory in repos
(cd $repos && git rm -r gamma)
(cd $repos && git commit -m "destroyed gamma")

# Modify the same directory in repos2, by adding a file to it
echo "iota" > $repos2/gamma/iota
# Make the modification visible in the index
(cd $repos2 && git add gamma/iota)
# Committing here does not affect the merge result.

# Pull the destruction of alpha into repos2 and merge
(cd $repos2 && git pull)
```

8.28 scripts/git/git-directories-create-created.sh

```
#!/bin/sh
# $Id: git-directories-create-created.sh 163 2008-08-12 14:31:30Z stsp $

. ./git-init.sh

# Create a directory in repos
mkdir $repos/eta
echo "theta" > $repos/eta/theta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "added eta")

# Create a different directory in repos2 at the same path
mkdir $repos2/eta
echo "iota" > $repos2/eta/iota
(cd $repos2 && git add eta)
# committing here does not affect the merge result

# Pull the add into repos2 and merge
(cd $repos2 && git pull)
```

8.29 scripts/git/git-directories-copy1-no-object.sh

```
#!/bin/sh
# $Id: git-directories-copy1-no-object.sh 175 2008-08-14 12:38:46Z stsp $

. ./git-init.sh

# Add a directory in repos
mkdir $repos/eta
echo "theta" > $repos/eta/theta
(cd $repos && git add eta)
(cd $repos && git commit -a -m "added eta")

# Create a copy of eta.
```

```
cp -r $repos/eta $repos/eta.copied
(cd $repos && git add eta.copied)
(cd $repos && git commit -a -m "copied eta")

# Cherry-pick just the copy into repos2
(cd $repos2 && git fetch)
rev=`(cd $repos2 && git log -1 --pretty=oneline origin | cut -d ' ' -f 1)`
(cd $repos2 && git cherry-pick $rev)
```

8.30 scripts/svn/svn-init.sh

```
#!/bin/sh
# $Id: svn-init.sh 186 2008-08-14 21:17:08Z stsp $

set -e

cwd=`pwd`
basename=`basename $0`
scratch_area="`echo $basename | sed -e s/\.sh$//`"
repos=$scratch_area/repos
trunk_url=file:///$cwd/$repos/trunk
branch_url=file:///$cwd/$repos/branch
trunk=$scratch_area/trunk
branch=$scratch_area/branch

rm -rf $scratch_area
mkdir -p $scratch_area

/bin/sh ../projtree.sh $trunk
svnadmin create $cwd/$repos
svn import $trunk $trunk_url -m "importing project tree"
svn copy $trunk_url $branch_url -m "creating branch"
rm -rf $trunk
svn checkout $trunk_url $trunk
svn checkout $branch_url $branch
```

8.31 scripts/svn/svn-files-move1-no-object.sh

```
#!/bin/sh
# $Id: svn-files-move1-no-object.sh 221 2008-08-24 13:36:45Z stsp $

. ./svn-init.sh

# Create a file in trunk
echo "eta" > $trunk/eta
svn add $trunk/eta
svn commit -m "added eta" $trunk

# Move the new file to another path in trunk
svn move $trunk/eta $trunk/eta.moved
svn commit -m "moved eta" $trunk
```

```
# Cherry-pick just the copy of eta into the branch
svn merge -c4 $trunk_url $branch
```

8.32 scripts/svn/svn-files-copy2-moved.sh

```
#!/bin/sh
# $Id: svn-files-copy2-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Copy a file in trunk
svn copy $trunk/alpha $trunk/alpha.copied
svn commit -m "copied alpha" $trunk

# In the branch, move a different file to the copy target path.
svn move $branch/beta $branch/alpha.copied
# Committing here does not affect the merge result

# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.33 scripts/svn/svn-directories-copy1-destroyed.sh

```
#!/bin/sh
# $Id: svn-directories-copy1-destroyed.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Copy a directory in trunk
svn copy $trunk/gamma $trunk/gamma.copied
svn commit -m "added gamma" $trunk

# Destroy the same directory in the branch
svn remove $branch/gamma
# Committing here does not affect the merge result

# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.34 scripts/svn/svn-files-modify-no-object.sh

```
#!/bin/sh
# $Id: svn-files-modify-no-object.sh 221 2008-08-24 13:36:45Z stsp $

. ./svn-init.sh

# Add a file in trunk
echo "eta" > $trunk/eta
svn add $trunk/eta
```

```
svn commit -m "added eta" $trunk

# Modify eta in trunk
echo "eta, modified" > $trunk/eta
svn commit -m "changed eta" $trunk

# Cherry-pick just the modification of eta into the branch
svn merge -c4 $trunk_url $branch
```

8.35 scripts/svn/svn-directories-move2-created.sh

```
#!/bin/sh
# $Id: svn-directories-move2-created.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Move a directory in trunk
svn move $trunk/gamma $trunk/gamma.moved
svn commit -m "moved gamma" $trunk

# In the branch, create a different directory at the move target path.
svn mkdir $branch/gamma.moved
echo "iota" > $branch/gamma.moved/iota
svn add $branch/gamma.moved/iota
# Committing here does not affect the merge result

# Merge the move into the branch
svn merge $trunk_url $branch
```

8.36 scripts/svn/svn-files-destroy-modified.sh

```
#!/bin/sh
# $Id: svn-files-destroy-modified.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Destroy the file eta on trunk
svn remove $trunk/alpha
svn commit -m "destroyed alpha" $trunk

# Modify the same file on the branch
echo "alpha, but modified" > $branch/alpha
# Committing here does affect the merge result,
# but no conflict is raised either way.
svn commit -m "modified alpha" $branch

# Merge from trunk into the branch
svn merge $trunk_url $branch
```

8.37 scripts/svn/svn-files-create-created.sh

```
#!/bin/sh
# $Id: svn-files-create-created.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Add a file in trunk
echo "eta" > $trunk/eta
svn add $trunk/eta
svn commit -m "added eta" $trunk

# Add a different file eta in the branch
echo "eta, but different" > $branch/eta
svn add $branch/eta
# Committing here does not affect the merge result

# Merge from trunk into the branch
svn merge $trunk_url $branch
```

8.38 scripts/svn/svn-files-copy1-no-object.sh

```
#!/bin/sh
# $Id: svn-files-copy1-no-object.sh 221 2008-08-24 13:36:45Z stsp $

. ./svn-init.sh

# Create a file in trunk
echo "eta" > $trunk/eta
svn add $trunk/eta
svn commit -m "added eta" $trunk

# Copy the new file to another path in trunk
svn copy $trunk/eta $trunk/eta.copied
svn commit -m "copied eta" $trunk

# Cherry-pick just the copy of eta into the branch
svn merge -c4 $trunk_url $branch
```

8.39 scripts/svn/svn-directories-move1-moved.sh

```
#!/bin/sh
# $Id: svn-directories-move1-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Move directory in trunk
svn move $trunk/gamma $trunk/gamma.moved1
svn commit -m "renamed gamma" $trunk

# Move the same directory to a different path in the branch
```

```
svn move $branch/gamma $branch/gamma.moved2
# Committing here does not affect the merge result

# Merge the move into the branch
svn merge $trunk_url $branch
```

8.40 scripts/svn/svn-directories-move2-moved.sh

```
#!/bin/sh
# $Id: svn-directories-move2-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Move a directory in trunk
svn move $trunk/gamma $trunk/gamma.moved
svn commit -m "moved gamma" $trunk

# In the branch, move a different directory to the move target path.
svn move $branch/epsilon $branch/gamma.moved
# Committing here does not affect the merge result

# Merge the move into the branch
svn merge $trunk_url $branch
```

8.41 scripts/svn/svn-directories-destroy-moved.sh

```
#!/bin/sh
# $Id: svn-directories-destroy-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Destroy the directory gamma on trunk
svn remove $trunk/gamma
svn commit -m "destroyed gamma" $trunk

# Move the same directory on the branch
svn move $branch/gamma $branch/gamma.moved
# Committing here does affect the merge result,
# but no conflict is raised either way.
svn commit -m "moved gamma" $branch

# Merge from trunk into the branch
svn merge $trunk_url $branch
```

8.42 scripts/svn/svn-directories-copy2-created.sh

```
#!/bin/sh
# $Id: svn-directories-copy2-created.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh
```

```
# Copy a directory in trunk
svn copy $trunk/gamma $trunk/gamma.copied
svn commit -m "copied gamma" $trunk

# In the branch, create a different directory at the copy target path.
svn mkdir $branch/gamma.copied
echo "iota" > $branch/gamma.copied/iota
svn add $branch/gamma.copied/iota
# Committing here does not affect the merge result

# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.43 scripts/svn/svn-files-create-moved.sh

```
#!/bin/sh
# $Id: svn-files-create-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Add a file in trunk
echo "eta" > $trunk/eta
svn add $trunk/eta
svn commit -m "added eta" $trunk

# In the branch, move a different file to the same path
svn move $branch/alpha $branch/eta
# Committing here does not affect the merge result

# Merge from trunk into the branch
svn merge $trunk_url $branch
```

8.44 scripts/svn/svn-directories-copy2-moved.sh

```
#!/bin/sh
# $Id: svn-directories-copy2-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Copy a directory in trunk
svn copy $trunk/gamma $trunk/gamma.copied
svn commit -m "copied gamma" $trunk

# In the branch, move a different directory to the copy target path.
svn move $branch/epsilon $branch/gamma.copied
# Committing here does not affect the merge result

# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.45 scripts/svn/svn-directories-move1-no-object.sh

```
#!/bin/sh
# $Id: svn-directories-move1-no-object.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Create directory in trunk
svn mkdir $trunk/eta
svn commit -m "added eta" $trunk

# Copy the new directory to another path in trunk
svn copy $trunk/eta $trunk/eta.copied
svn commit -m "copied eta" $trunk

# Cherry-pick just the move into the branch
svn merge -c4 $trunk_url $branch
```

8.46 scripts/svn/svn-directories-destroy-modified.sh

```
#!/bin/sh
# $Id: svn-directories-destroy-modified.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Destroy the directory gamma on trunk
svn remove $trunk/gamma
svn commit -m "destroyed gamma" $trunk

# Modify the same directory on the branch
echo "iota" > $branch/gamma/iota
svn add $branch/gamma/iota
# Committing here does affect the merge result,
# but no conflict is raised either way.
svn commit -m "modified gamma" $branch

# Merge from trunk into the branch
svn merge $trunk_url $branch
```

8.47 scripts/svn/svn-directories-create-created.sh

```
#!/bin/sh
# $Id: svn-directories-create-created.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Add a directory eta in trunk
svn mkdir $trunk/eta
echo "iota" > $trunk/eta/iota
svn add $trunk/eta/iota
svn commit -m "added eta" $trunk
```

```
# Add a different directory in the branch
svn mkdir $branch/eta
echo "theta" > $branch/eta/theta
svn add $branch/eta/theta
# Committing here does not affect the merge result

# Merge from trunk into the branch
svn merge $trunk_url $branch
```

8.48 scripts/svn/svn-directories-copy1-no-object.sh

```
#!/bin/sh
# $Id: svn-directories-copy1-no-object.sh 221 2008-08-24 13:36:45Z stsp $

. ./svn-init.sh

# Create directory in trunk
svn mkdir $trunk/eta
svn commit -m "added eta" $trunk

# Copy the new directory to another path in trunk
svn copy $trunk/eta $trunk/eta.copied
svn commit -m "copied eta" $trunk

# Cherry-pick just the copy of eta into the branch
svn merge -c4 $trunk_url $branch
```

8.49 scripts/svn/svn-directories-create-moved.sh

```
#!/bin/sh
# $Id: svn-directories-create-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Add a directory eta in trunk
svn mkdir $trunk/eta
echo "iota" > $trunk/eta/iota
svn add $trunk/eta/iota
svn commit -m "added eta" $trunk

# In the branch, move a different directory to the move target path
svn move $branch/gamma $branch/eta
# Committing here does not affect the merge result

# Merge from trunk into the branch
svn merge $trunk_url $branch
```

8.50 scripts/svn/svn-files-modify-destroyed.sh

```
#!/bin/sh
# $Id: svn-files-modify-destroyed.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Modify a file in trunk
echo "alpha, modified" > $trunk/alpha
svn commit -m "changed alpha" $trunk

# Destroy alpha in the branch
svn remove $branch/alpha
# Committing here does not affect the merge result

# Merge the change into the branch
svn merge $trunk_url $branch
```

8.51 scripts/svn/svn-files-move1-destroyed.sh

```
#!/bin/sh
# $Id: svn-files-move1-destroyed.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Move a file in trunk
svn move $trunk/alpha $trunk/alpha.moved
svn commit -m "moved alpha" $trunk

# Destroy the same file in the branch
svn remove $branch/alpha
# Committing here does not affect the merge result

# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.52 scripts/svn/svn-files-copy1-destroyed.sh

```
#!/bin/sh
# $Id: svn-files-copy1-destroyed.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Copy a file in trunk
svn copy $trunk/alpha $trunk/alpha.copied
svn commit -m "added alpha" $trunk

# Destroy the same file in the branch
svn remove $branch/alpha
# Committing here does not affect the merge result
```

```
# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.53 scripts/svn/svn-files-move2-created.sh

```
#!/bin/sh
# $Id: svn-files-move2-created.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Move a file in trunk
svn move $trunk/alpha $trunk/alpha.moved
svn commit -m "moved alpha" $trunk

# In the branch, create a different file at the move target path.
echo "alpha.moved, different" > $branch/alpha.moved
svn add $branch/alpha.moved
# Committing here does not affect the merge result

# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.54 scripts/svn/svn-files-move1-moved.sh

```
#!/bin/sh
# $Id: svn-files-move1-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Move a file in trunk
svn move $trunk/alpha $trunk/alpha.moved1
svn commit -m "moved alpha" $trunk

# Move the same file to a different path in the branch
svn move $branch/alpha $branch/alpha.moved2
# Committing here does not affect the merge result

# Merge the rename into the branch
svn merge $trunk_url $branch
```

8.55 scripts/svn/svn-files-move2-moved.sh

```
#!/bin/sh
# $Id: svn-files-move2-moved.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Move a file in trunk
svn move $trunk/alpha $trunk/alpha.moved
svn commit -m "moved alpha" $trunk
```

```
# In the branch, move a different file to the move target path.
svn move $branch/beta $branch/alpha.moved
# Committing here does not affect the merge result

# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.56 scripts/svn/svn-files-copy2-created.sh

```
#!/bin/sh
# $Id: svn-files-copy2-created.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Copy a file in trunk
svn copy $trunk/alpha $trunk/alpha.copied
svn commit -m "copied alpha" $trunk

# In the branch, create a different file at the copy target path.
echo "alpha.copied, different" > $branch/alpha.copied
svn add $branch/alpha.copied
# Committing here does not affect the merge result

# Merge the copy into the branch
svn merge $trunk_url $branch
```

8.57 scripts/svn/svn-directories-move1-destroyed.sh

```
#!/bin/sh
# $Id: svn-directories-move1-destroyed.sh 227 2008-08-24 14:39:21Z stsp $

. ./svn-init.sh

# Move a directory in trunk
svn move $trunk/gamma $trunk/gamma.moved
svn commit -m "moved gamma" $trunk

# Destroy the same directory in the branch
svn remove $branch/gamma
# Committing here does not affect the merge result

# Merge the move into the branch
svn merge $trunk_url $branch
```

8.58 scripts/svn/svn-files-destroy-moved.sh

```
#!/bin/sh
# $Id: svn-files-destroy-moved.sh 227 2008-08-24 14:39:21Z stsp $
```

```
. ./svn-init.sh

# Destroy a file on trunk
svn remove $trunk/alpha
svn commit -m "destroyed alpha" $trunk

# Rename the same file on the branch
svn move $branch/alpha $branch/alpha.moved
# Committing here does not affect the merge result

# Merge from trunk into the branch
svn merge $trunk_url $branch
```

8.59 scripts/hg/hg-directories-destroy-modified.sh

```
#!/bin/sh
# $Id: hg-directories-destroy-modified.sh 119 2008-08-05 11:30:08Z stsp $

. ./hg-init.sh

# Remove a directory from repos
hg remove $repos/gamma
hg commit -m "removed gamma" $repos

# Modify the same directory in repos2's working copy
echo "theta" > $repos2/gamma/theta
hg add $repos2/gamma/theta

# Pull the delete into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.60 scripts/hg/hg-directories-move2-moved.sh

```
#!/bin/sh
# $Id: hg-directories-move2-moved.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Move a directory in repos
hg rename $repos/gamma $repos/gamma.moved
hg commit -m "renamed gamma" $repos

# Move a different directory to the move target path in repos2
hg rename $repos2/epsilon $repos2/gamma.moved

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.61 scripts/hg/hg-directories-destroy-moved.sh

```
#!/bin/sh
# $Id: hg-directories-destroy-moved.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Remove a directory from repos
hg remove $repos/gamma
hg commit -m "removed gamma" $repos

# Rename the same directory in repos2's working copy
hg rename $repos2/gamma $repos2/gamma.moved

# Pull the delete into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.62 scripts/hg/hg-directories-copy2-created.sh

```
#!/bin/sh
# $Id: hg-directories-copy2-created.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Copy a directory in repos
hg copy $repos/gamma $repos/gamma.copied
hg commit -m "copied gamma" $repos

# Create a different directory in repos2 at the same path
mkdir $repos2/gamma.copied
echo "eta" > $repos2/gamma.copied/eta
hg add $repos2/gamma.copied

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.63 scripts/hg/hg-files-create-moved.sh

```
#!/bin/sh
# $Id: hg-files-create-moved.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Create a file in repos
echo "eta" > $repos/eta
hg commit -A -m "added eta" $repos

# Move a different file to the same path in repos2
hg rename $repos2/alpha $repos2/eta

# Pull the add into repos2 and update
```

```
(cd $repos2 && hg pull; hg update)
```

8.64 scripts/hg/hg-directories-copy-moved-to.sh

```
#!/bin/sh
# $Id: hg-directories-copy-moved-to.sh 100 2008-07-23 14:13:57Z stsp $

. ./hg-init.sh

# Copy a directory in repos
hg copy $repos/gamma $repos/gamma.copied
hg commit -m "copied gamma" $repos

# Move the same directory in repos2's working copy
hg rename $repos2/gamma $repos2/gamma.moved

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.65 scripts/hg/hg-directories-copy2-moved.sh

```
#!/bin/sh
# $Id: hg-directories-copy2-moved.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Copy a directory in repos
hg copy $repos/gamma $repos/gamma.copied
hg commit -m "copied gamma" $repos

# Move a different directory in repos2 to the copy target path
hg rename $repos2/epsilon $repos2/gamma.copied

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.66 scripts/hg/hg-files-move-moved-from.sh

```
#!/bin/sh
# $Id: hg-files-move-moved-from.sh 100 2008-07-23 14:13:57Z stsp $

. ./hg-init.sh

# Move alpha into gamma/ in repos
hg rename $repos/alpha $repos/gamma/
hg commit -m "moved alpha" $repos

# Rename beta to alpha in repos2.
hg rename --force $repos2/beta $repos2/alpha
```

```
# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.67 scripts/hg/hg-directories-move1-destroyed.sh

```
#!/bin/sh
# $Id: hg-directories-move1-destroyed.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Move a directory in repos
hg rename $repos/gamma $repos/gamma.moved
hg commit -m "moved gamma" $repos

# Delete the same directory in repos2's working copy
hg remove $repos2/gamma

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.68 scripts/hg/hg-directories-create-created.sh

```
#!/bin/sh
# $Id: hg-directories-create-created.sh 119 2008-08-05 11:30:08Z stsp $

. ./hg-init.sh

# Add a directory in repos
mkdir $repos/eta
echo "theta" > $repos/eta/theta
hg commit -A -m "added eta/" $repos

# Add a directory with the same name but different content
# in repos2's working copy
mkdir $repos2/eta
echo "iota" > $repos2/eta/iota
hg add $repos2/eta

# Pull the add into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.69 scripts/hg/hg-directories-create-moved.sh

```
#!/bin/sh
# $Id: hg-directories-create-moved.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Add a directory in repos
mkdir $repos/eta
```

```
echo "theta" > $repos/eta/theta
hg commit -A -m "added eta/" $repos

# Move a different directory to the same path in repos2
hg rename $repos2/gamma $repos2/eta

# Pull the add into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.70 scripts/hg/hg-files-modify-destroyed.sh

```
#!/bin/sh
# $Id: hg-files-modify-destroyed.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Modify a file in repos
echo "alpha, modified" > $repos/alpha
hg commit -m "changed alpha" $repos

# Destroy the same file in repos2's working copy
hg remove $repos2/alpha

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.71 scripts/hg/hg-files-move1-destroyed.sh

```
#!/bin/sh
# $Id: hg-files-move1-destroyed.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Move a file in repos
hg rename $repos/alpha $repos/alpha.moved
hg commit -m "moved alpha" $repos

# Delete the same file in repos2's working copy
hg remove $repos2/alpha

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.72 scripts/hg/hg-files-copy1-destroyed.sh

```
#!/bin/sh
# $Id: hg-files-copy1-destroyed.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh
```

```
# Copy a file in repos
hg copy $repos/alpha $repos/alpha.copied
hg commit -m "copied alpha" $repos

# Delete the same file in repos2's working copy
hg remove $repos2/alpha

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.73 scripts/hg/hg-files-move2-created.sh

```
#!/bin/sh
# $Id: hg-files-move2-created.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Move a file in repos
hg rename $repos/alpha $repos/eta
hg commit -m "renamed eta" $repos

# Create a file at the move target path in repos2
echo "eta, but different" > $repos2/eta
hg add $repos2/eta

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.74 scripts/hg/hg-files-destroy-created.sh

```
#!/bin/sh
# $Id: hg-files-destroy-created.sh 119 2008-08-05 11:30:08Z stsp $

. ./hg-init.sh

# Remove a file from repos
hg remove $repos/alpha
hg commit -m "removed alpha" $repos

# Remove the same file from repos2, and locally add it again.
hg remove $repos2/alpha
hg commit -m "removed alpha, too" $repos2
echo "alpha, resurrected" >> $repos2/alpha
hg add $repos2/alpha

# Pull the delete into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.75 scripts/hg/hg-files-move1-moved.sh

```
#!/bin/sh
# $Id: hg-files-move1-moved.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Move a file in repos
hg rename $repos/alpha $repos/alpha.moved
hg commit -m "moved alpha to alpha.moved" $repos

# Move the same file to a different location in repos2's working copy
hg rename $repos2/alpha $repos2/alpha.moved2

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.76 scripts/hg/hg-files-move2-moved.sh

```
#!/bin/sh
# $Id: hg-files-move2-moved.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Move a file in repos
hg rename $repos/alpha $repos/eta
hg commit -m "renamed eta" $repos

# Move another file to the move target path in repos2
hg rename $repos2/beta $repos2/eta

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.77 scripts/hg/hg-files-copy2-created.sh

```
#!/bin/sh
# $Id: hg-files-copy2-created.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Copy a file to repos
hg copy $repos/alpha $repos/eta
hg commit -m "copied alpha to eta" $repos

# Add a different file at the copy target path in repos2
echo "eta" > $repos2/eta
hg add $repos2/eta

# Pull the copy into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.78 scripts/hg/hg-directories-move2-created.sh

```
#!/bin/sh
# $Id: hg-directories-move2-created.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Move a directory in repos
hg rename $repos/gamma $repos/gamma.moved
hg commit -m "renamed gamma" $repos

# Create a different directory at the move target path in repos2
mkdir $repos2/gamma.moved
echo "eta" > $repos2/gamma.moved/eta
hg add $repos2/gamma.moved/eta

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.79 scripts/hg/hg-files-destroy-moved.sh

```
#!/bin/sh
# $Id: hg-files-destroy-moved.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Remove a file from repos
hg remove $repos/alpha
hg commit -m "removed alpha" $repos

# Rename the same file in repos2's working copy
hg rename $repos2/alpha $repos2/alpha.moved

# Pull the delete into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.80 scripts/hg/hg-init.sh

```
#!/bin/sh
# $Id: hg-init.sh 100 2008-07-23 14:13:57Z stsp $

set -e

basename=‘basename $0‘
scratch_area="‘echo $basename | sed -e s/\.sh$//‘"
repos=$scratch_area/repos
repos2=$scratch_area/repos2

rm -rf $scratch_area

/bin/sh ../projtree.sh $repos
```

```
hg init $repos
(cd $repos && hg commit -A -m "importing project tree")
hg clone $repos $repos2
(cd $repos2 && hg update)
```

8.81 scripts/hg/hg-files-copy2-moved.sh

```
#!/bin/sh
# $Id: hg-files-copy2-moved.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Copy a file in repos
hg copy $repos/alpha $repos/alpha.copied
hg commit -m "copied alpha" $repos

# Move another file in repos2 to the copy target path
hg rename $repos2/beta $repos2/alpha.copied

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.82 scripts/hg/hg-directories-copy1-destroyed.sh

```
#!/bin/sh
# $Id: hg-directories-copy1-destroyed.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Copy a directory in repos
hg copy $repos/gamma $repos/gamma.copied
hg commit -m "copied gamma" $repos

# Delete the same directory in repos2's working copy
hg remove $repos2/gamma

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.83 scripts/hg/hg-files-destroy-modified.sh

```
#!/bin/sh
# $Id: hg-files-destroy-modified.sh 119 2008-08-05 11:30:08Z stsp $

. ./hg-init.sh

# Remove a file from repos
hg remove $repos/alpha
hg commit -m "removed alpha" $repos
```

```
# Modify the same file in repos2's working copy
echo "alpha, modified" > $repos2/alpha

# Pull the delete into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.84 scripts/hg/hg-files-create-created.sh

```
#!/bin/sh
# $Id: hg-files-create-created.sh 140 2008-08-09 16:22:25Z stsp $

. ./hg-init.sh

# Create a file in repos
echo "eta" > $repos/eta
hg commit -A -m "added eta" $repos

# Create a file with the same name but different content
# in repos2's working copy
echo "eta, but different" > $repos2/eta
hg add $repos2/eta

# Pull the add into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.85 scripts/hg/hg-directories-move1-moved.sh

```
#!/bin/sh
# $Id: hg-directories-move1-moved.sh 142 2008-08-10 19:41:51Z stsp $

. ./hg-init.sh

# Move a directory in repos
hg rename $repos/gamma $repos/gamma.moved
hg commit -m "moved gamma to gamma.moved" $repos

# Move the same directory to a different location in repos2's working copy
hg rename $repos2/gamma $repos2/gamma.moved2

# Pull the modification into repos2 and update
(cd $repos2 && hg pull; hg update)
```

8.86 scripts/projtree.sh

```
#!/bin/sh
# $Id: projtree.sh 79 2008-06-16 15:21:49Z stsp $

if [ $# != 1 ]; then
        echo -n "Creates a new directory and populates it with "
```

```
        echo "a pseudo project tree."
        echo "Usage: `basename $0` <directory>"
        exit 1
fi

if [ -e $1 ]; then
        echo "ERROR: $1 already exists!" 1>&2
        exit 1
fi

set -e

mkdir -p $1
echo alpha > $1/alpha
echo beta > $1/beta
mkdir $1/gamma
echo delta > $1/gamma/delta
mkdir $1/epsilon
echo zeta > $1/epsilon/zeta
```