A Principled Approach to Version Control

Andres Löh[1], Wouter Swierstra[2], and Daan Leijen[3]

[1] University of Bonn
loeh@informatik.uni-bonn.de
[2] University of Nottingham
wss@cs.nott.ac.uk
[3] Microsoft Research
daan@microsoft.com

Abstract. Version control systems are essential for managing the distributed development of large software projects. We present a formal model for reasoning about version control. In particular, we give a general definition of *patch*. Patches abstract over the data on which they operate, making our framework equally suited for version control of everything from highly-structured XML files to blobs of bits. We model repositories as a multiset of patches. The mathematical definitions of patches and repositories enable us to reason about complicated issues such as conflicts and conflict resolution.

1 Introduction

Version control systems tackle a real problem: how should we manage distributed development? Modern software projects frequently require multiple developers, based at different locations, editing thousands of files simultaneously. Manual revision control is simply no longer an option.

Early version control systems were often simple tools capable of tracking the revisions of a single text file. In the last twenty years, however, version control systems have evolved into complex programs managing huge source trees with multiple branches.

Despite this rapid development, version control systems have remained rather ad-hoc. Tools are designed and refined to solve perceived problems with little regard for fundamental issues. As a result, programmers dread performing complex operations on a repository, such as merging branches or resolving conflicts. These operations can be both unwieldy and unpredictable.

This paper aims to give a precise, mathematical description of the version control problem and different version control systems. We introduce the problem by formalizing a simple version control system for unstructured binary files (Section 2). We generalize this example and give a precise definition of patches and repositories (Section 3). Building on this, we introduce a few extensions to our core model such as directory structure, tagged versions, and patch metadata (Section 4). Using our model, we can accurately predict when conflicts may arise and how they may be resolved (Section 5). We believe that this work may

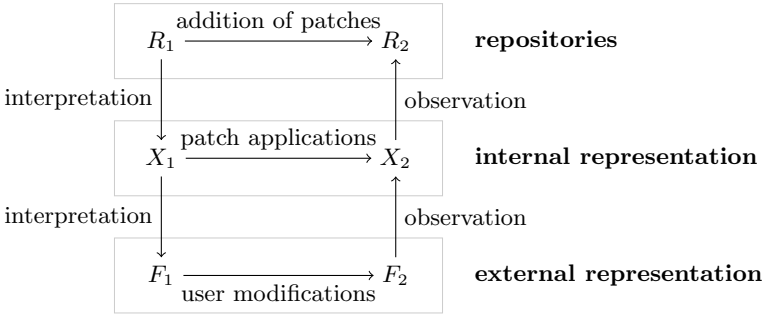

Fig. 1. Representation of repositories

lay the groundwork for reasoning about version control, and ultimately come up with better and more predictable systems.

Terminology

A version control system manages *repositories*. A repository can consist of anything from single file to a complete source tree. In addition to this current representation of its state, a repository contains *history*. Each repository's internal state is the result of several *patches*. A patch is a logically connected collection of operations that changes the state of a repository.

The state of a repository exists on two different levels. We shall refer to the actual files and directories on disk as the *external representation*. Version control systems also have an *internal representation*, or abstract model capturing information of interest about the files in the repository.

Of course, there is a relationship between a repository and its internal and external representation. When a user changes a file, that change must percolate into a system's internal representation and ultimately result in the generation of a new patch – a process we shall refer to as *observation*. Conversely, when a user pulls in changes from another repository, these patches must somehow be *interpreted* as changes to the files on disk. The internal representation typically fulfills certain *invariants* that ensure that it can indeed be interpreted in the file system. This relationship is shown in Figure 1. In this paper, we will not deal with the external representation or observation. Instead, we focus on the internal representation and how to apply patches. The interpretation of the internal representations we give is usually obvious. Observation, on the other hand, will usually require tool support: a user's changes to the file system cannot always be uniquely associated with a single patch. In such cases, the user might be asked to provide additional information to help the tool produce the correct patches.

One important selling point of our theory is that it abstracts over internal representations. Different version control systems serve different purposes: some manage source files on a line-by-line basis; others manage structured XML files;

others yet again may simply manage binary data. Any underlying theory must be capable of dealing with each and every one of these potential design choices.

Patches can be moved between repositories, but this may lead to *conflicts*. Sometimes, a move of one patch will imply the move of several dependent patches in order to prevent a conflict. On other occasions, resolving conflicts will require user interaction.

Many other notions usually found in version control systems are not part of our core model. We do not distinguish between centralized and distributed version control systems. We also do not have any specific treatment of branches or even projects. In principle, any patch can be moved freely between any two repositories; heedlessly doing so will often result in conflicts. Some version control systems may maintain additional policies to relate specific repositories. Such policies are specific to a version control system and by no means fundamental to any theory of version control.

2 Managing binary files

To illustrate the concepts of our theory, we give a precise description of a simple version control system capable of managing binary files. We do not pretend that this system will be of much practical use, but it nicely illustrates the fundamental issues involved. In later sections, we will extend it incrementally into a full-fledged system handling text files, directory structure and file renames.

If we want to formalize such a version control system, we must begin by defining what a repository is. This is an important question: should we really construct a complete formal model of files, directories, permissions, and ownership? Clearly it would better to build a more abstract internal representation that captures the information we are interested in, but no more than that. This motivates the following model of repositories.

2.1 Repository

We assume there is a set F of all valid file names. We consider the contents of a file to be simply a sequence of bits, simply denoted by Bits. We write empty to denote the empty bit sequence. We use a single predicate to state information about our repository: $f = c$ states that the file the file f has contents some sequence of bits c. We can now model the state of the repository (its internal representation) as a set of such predicates.

Not every set of predicates forms a valid repository. We require that the following invariant holds for any internal representation X:

$$\forall(f \in F).\exists_{\leqslant 1}(c \in \mathit{Bits}).(f = c) \in X \quad .$$

This invariant ensures that every file that exists has uniquely determined contents. A newly created repository is represented by an empty set.

2.2 Operations on the repository

Now we have fixed our notion of repository, we can begin to define operations that somehow change the repository. In particular, we consider the following three operations on repositories:

- adding an empty file to the repository,
- removing an existing file from the repository,
- and updating the contents of a file in the repository.

We can now define how these operations affect the internal representation of the repository:

$$\begin{aligned} \mathit{add}\ f\ r &= r \cup \{f = \mathsf{empty}\} \\ \mathit{remove}\ f\ c\ r &= r \setminus \{f = c\} \\ \mathit{replace}\ f\ c\ d\ r &= (r \setminus \{f = c\}) \cup \{f = d\}\ . \end{aligned}$$

There are, however, some real problems with blindly applying these functions to a repository. The behavior of a version control system can become a bit surprising. For instance, suppose we try to add a file to a repository that already exists. The result of doing so, would break the repository invariants we described above! Similarly, deleting a file that does not exist, does not change anything – yet somehow the user should be warned that such an operation is a bit dubious. Clearly some patches do not make sense in every repository. This motivates a more advanced model of patches.

A *patch* $p = S \vdash E \rightarrow T$ consists of a triple of sets S, E, and T. We shall refer to S as the *source* of p and T as its *target*. The notation for patches is intended to suggest an interrupted arrow from S to T carrying additional information. Intuitively, such a patch removes the source S from the repository and adds the target T – hence the arrow-like notation. To apply such a patch to a repository, the set S must already be present in the repository; conversely, the set $T \setminus S$ must be absent – the patch should not try to add data that is present after removing S.

The set E, called the *extension* of p, is a bit more subtle. We require E to be a superset of S and T – in many cases, $S \cup T$ will in fact be equal to E. The extension, however, may contain points beyond S and T that are somehow affected by the patch. Such points must be absent in the representation when we apply the patch, and are guaranteed to be absent after the application. They might be required to ensure that applying the patch will maintain the repository invariants, or the might reserve space for temporary values that are created, but also removed again by the patch. We shall give a more precise definition of a patch in Section 3. For now, we use this informal definition to give a better definition of our operations.

Creating files A file with name $f \in F$ can be created using a patch:

$$\mathit{create}\ f = \varnothing \vdash \{f = c \mid c \in \mathit{Bits}\} \rightarrow \{f = \mathsf{empty}\}\ .$$

The target of the resulting patch indicates that files are always initially created without contents. The source of the patch is the empty set: creation of files does not rely on any current contents of the repository. It is, however, important that the file we are about to create does not yet exist. This is ensured by the extension: remember that any point in the extension outside the source must be absent from the repository to apply a patch. Because files cannot be added twice, the invariant of the internal representation is maintained by the patch.

We will occasionally abbreviate sets using an asterisk wild-card. When the type of a variable can be inferred from the context, we may write for instance

$$\mathit{create}\ f \qquad = \varnothing \vdash \{f = *\} \rightarrowtail \{f = \mathsf{empty}\}\ .$$

Different occurrences of $*$ within a set are independent.

Deleting files We can delete a file as follows:

$$\mathit{delete}\ f\ c \qquad = \{f = c\} \vdash \{f = c\} \rightarrowtail \varnothing\ .$$

Note that *delete* is parameterized over the final contents of the file c. We require the file f to exist prior to removal.

This operation exemplifies a common situation: the extension does not have points outside the source and target. In that case, we will sometimes write patches using the following shorthand:

$$\mathit{delete}\ f\ c \qquad = \{f = c\} \mapsto \varnothing\ .$$

We omit the extension, as it can be inferred from the source and target.

Modifying files We can modify the contents of an existing file using:

$$\mathit{modify}\ f\ c\ d = \{f = c\} \mapsto \{f = d\}\ .$$

Both the old and the new contents of the binary file are parameters of the patch.

It is easy to see that patches given by *create*, *delete* and *modify* maintain the invariant of the internal representation that we specified above.

We conclude this section with one last example. We create a file `"foo"`, set its contents to the bit sequence 010, and remove it again, with the following sequence of patches:

$$\mathit{create}\ \texttt{"foo"}, \mathit{modify}\ \texttt{"foo"}\ \mathsf{empty}\ 010, \mathit{delete}\ \texttt{"foo"}\ 010\ .$$

The internal representation resulting from these three patches is the empty set again. A repository will record and remember all patches that are applied to it, thereby recording sufficient information to reproduce its entire history.

Consider Figure 1 again. Although we have discussed the top two layers of that diagram, we have not mentioned the external representation. An implementation of a version control system must map the internal representation to an actual file system; conversely, it must also observe changes made by the user

to files in terms of the internal representation, and ultimately in terms of new patches. For instance, if the user creates another file `"bar"`, the system should observe the change by adding a patch *create* `"bar"`.

There are several variations and extensions of the patches defined above. We will discuss many additional design choices in Section 4.

3 Formal model

While the management binary files makes an interesting example, the notions we used still need to be defined rigorously. This section presents the core definitions of patches, patch application, and patch composition that lie at the heart of our theory.

Definition 1 (patch). A **patch**

$$f = S \vdash E \rightarrow T$$

is a triple of sets, S, E and T, where $S \subseteq E$ and $T \subseteq E$. We say that $\mathsf{src}\ f = S$ is the **source** of f and that $\mathsf{tgt}\ f = T$ is the **target** of f. Finally, we refer to $\mathsf{ext}\ f = E$ as the **extension** of f.

For each set X with $X \cap E = S$, we define:

$$f\ (X) := S \triangle X \triangle T$$

and say that f is **applicable** to X and that $f\ (X)$ is the result of **applying** f to X. We write $S \mapsto T$ if $E = S \cup T$.

A few remarks are in order. The operation $\triangle$ denotes the symmetric difference of two sets, i.e., $A \triangle B = (A \setminus B) \cup (B \setminus A)$. We could have defined patch application equivalently as $f\ (X) := (X \setminus S) \cup T$, but symmetric difference has nicer properties in calculations.

For a patch $S \vdash E \rightarrow T$ We do not require $S \cap T = \varnothing$. Sometimes, a patch requires something to be present in the representation, but does not modify or remove it. An example is creating a file in a hierarchical directory structure as shown in Section 4.2, where the parent directory is required to exist before and after the application of the patch.

Patches in isolation are not terribly useful. Now we have given a formal definition of patches, we can define how individual patches can be composed.

Definition 2 (composition of patches). Two patches $f = S_1 \vdash E_1 \rightarrow T_1$ and $g = S_2 \vdash E_2 \rightarrow T_2$ can be **composed** if and only if

$$(E_1 \cap (S_2 \setminus T_1)) = ((T_1 \setminus S_2) \cap E_2) = \varnothing \ .$$

We then define

$$g \cdot f := (S_1 \cup (S_2 \setminus T_1)) \vdash (E_1 \cup E_2) \rightarrow ((T_1 \setminus S_2) \cup T_2)$$

to be the **composition** of f and g.

It is easy to see that $g \cdot f$ is a well-defined patch. For instance, we need to check that $(S_1 \cup (S_2 \setminus T_1)) \subseteq E_1 \cup E_2$ – but this follows immediately from the fact that f and g are well-defined patches. Our functional notation of patches is purposefully suggestive. The following is special case of function composition:

Corollary 3. *Any patches of the form $f = S \vdash E_1 \rightarrow T$ and $g = T \vdash E_2 \rightarrow U$ are composable and $g \cdot f = S \vdash (E_1 \cup E_2) \rightarrow U$. In particular, if $f = S \mapsto T$ and $g = T \mapsto U$, then $g \cdot f = S \vdash S \cup T \cup U \rightarrow U$.*

The following results show that our definition of composition behaves more or less like function composition.

Lemma 4. *Suppose $f = S_1 \vdash E_1 \rightarrow T_1$ and $g = S_2 \vdash E_2 \rightarrow T_2$ are composable patches and $g \cdot f$ is applicable to a set X. Then f is applicable to X, and g is applicable to $f\ (X)$, and $(g \cdot f)\ (X) = g\ (f\ (X))$.*

The proof is not very deep, but fairly technical; we have included it in Appendix A for the sake of completeness.

Composition is associative and even commutative under suitable conditions. These results are not only of theoretical interest: we will make grateful use of them in our exposition on repositories and conflicts.

Corollary 5. *For any suitably composable patches f, g, and h, we can show that $f \cdot (g \cdot h) = (f \cdot g) \cdot h$. Put succinctly, composition is associative.*

Similarly, if patches f and g are composable in both directions and $g \cdot f$ and $f \cdot g$ are both applicable to X, then $(g \cdot f)\ (X) = (f \cdot g)\ (X)$. In other words, patch composition is commutative for sufficiently composable patches.

Proof. Both these properties follow immediately from the associativity and commutativity of symmetric difference. □

Finally, we have a notion of inverse and identity patch. These patches are also of practical use, as we shall see in the coming sections.

Definition 6 (inverse and identity of patches). For any patch $f = S \vdash E \rightarrow T$, we call $\mathsf{inv}\ f := T \vdash E \rightarrow S$ the **inverse** of f. A patch $\mathsf{id}_{A,E} = A \vdash E \rightarrow A$ is called an **identity patch**. We write id_A for $A \mapsto A$.

Lemma 7. *For any patch $f = S \vdash E \rightarrow T$, we have*

$$\mathsf{inv}\ f \cdot f = \mathsf{id}_{S,E} \quad \textit{and} \quad f \cdot \mathsf{inv}\ f = \mathsf{id}_{T,E}\ .$$

Proof. Obvious from the definitions of inverse and composition. □

4 Beyond binary files

The previous section covered the mathematical core of our theory. Before we continue down that road, let us try to apply what we have so far. In this section, we present several concepts and examples that extend upon the simple version control for binary files we introduced in Section 2, and explain how all of them can be expressed in terms of patches.

4.1 Renaming files

So far we have considered binary files with fixed names. We could create, modify, and remove them. If we wanted to transfer the contents of one file to another, we would have to create a new file and set its contents to that of the old file. But by then, all connection between the two files is lost.

We can do better and assign names to files in such a way that files can be renamed easily. We store the name of a file in the repository using an entry of the form f is n. The function

$$\mathit{create}\ f\ n = \varnothing \vdash \{f = *, * \text{ is } n\} \rightarrow \{f = \mathsf{empty}, f \text{ is } n\}$$

results in a patch that creates the file f with initial name n. Recall that $\{f = *, * \text{ is } n\}$ serves as an abbreviation for the set $\{f = c \mid c \in \mathit{Bits}\} \cup \{g \text{ is } n \mid g \in F\}$.

Note that f is now a purely internal label that identifies the file uniquely, whereas n is an externally visible name of the file that can be modified by the user at any time. As such, the name is not much different from the contents: it is a property of the file that can be changed. Indeed, changing the name of a file is achieved using a function very similar to the *modify* function:

$$\mathit{rename}\ f\ n_1\ n_2 = \{f \text{ is } n_1\} \mapsto \{f \text{ is } n_2\}\ .$$

Most of the time when we introduce new types of patches we have to adapt the invariants of our internal representation. We require that every file has just one name and that no two files of the same name occur in a representation X:

$$\forall (f \in F).\exists_{\leqslant 1}(n \in \mathit{Names}).\{f \text{ is } n\} \in X$$
$$\forall (n \in \mathit{Names}).\exists_{\leqslant 1}(f \in F).\{f \text{ is } n\} \in X\ .$$

4.2 Directory hierarchy and file moves

The next step is to add a hierarchical directory structure to our repository. We treat directories as special files, and write $f = \mathsf{dir}$ to indicate that the label f refers to a directory rather than to a regular file. Let $\mathit{DBits} = \mathit{Bit} \cup \{\mathsf{dir}\}$. If we write $\{f = *\}$, it is now an abbreviation for $\{f = c \mid c \in \mathit{DBits}\}$ Directory creation is then fairly straightforward:

$$\begin{aligned}
&\mathit{create}\ f\ n\ c &&= \varnothing \vdash \{f = *, * \text{ is } n\} \rightarrow \{f = c, f \text{ is } n\}\\
&\mathit{createDir}\ f\ n &&= \mathit{create}\ f\ n\ \mathsf{dir}\\
&\mathit{createFile}\ f\ n &&= \mathit{create}\ f\ n\ \mathsf{empty}\ .
\end{aligned}$$

Directory creation is like file creation, only that we mark the file as a directory immediately rather than to assign empty contents.

Unfortunately, there is no way to store contents in these directories yet. To add tree structure, we extend the is predicate to read f is $d\,/\,n$, associating with each file not only a name n, but also the directory label d where it is located.

In particular, we can model nested directories, as directory and file labels share the same namespace.

Everything is now created within a specific directory, hence we refine *create*:

$$\begin{aligned}&\mathit{create}\ f\ d\ n\ c = \\ &\quad \{\, d = \mathsf{dir} \,\} \vdash \{ f = *, * \ \mathsf{is}\ d\ /\ n, d = \mathsf{dir} \} \rightarrow \{\, d = \mathsf{dir}, f = c, f\ \mathsf{is}\ d\ /\ n \,\}\ .\end{aligned}$$

Creating a new file requires the target directory d to exist already. Where do we place the initial file? We assume that a special root directory is in the initial repository via an entry $\mathsf{root} = \mathsf{dir}$. The name and the location of the root directory are immutable.

We can move a file to a new location and possibly a new name using:

$$\mathit{move}\ f\ d_1\ n_1\ d_2\ n_2 = f\ \mathsf{is}\ d_1\ /\ n_1 \mapsto f\ \mathsf{is}\ d_2\ /\ n_2\ .$$

Of course, we also need a whole bunch of new or modified invariants. File names must now only be unique within a directory. A file can be a directory or a regular file, but not both. The parent of a file is always a directory. Every file but root has a location and a name, and the location must be an existing directory. Due to the last invariant, we must make sure that we only delete empty directories:

$$\mathit{removeDir}\ d = \{\, d = \mathsf{dir} \,\} \vdash \{ * \ \mathsf{is}\ d\ /\ *, d = \mathsf{dir} \} \rightarrow \varnothing\ .$$

4.3 Line-based text files

Most version control systems are targeted at text files. They typically allow operations such as the insertion or the deletion of text on a line-by-line basis.

In contrast to most version control systems, we choose not to deal with absolute line numbers. Instead we assign internal labels to lines, which we chain together as a linked list. This is actually quite similar to how we have dealt with file names and directories. We try to specify the intent of a patch as closely as possible in its definition.

For the sake of simplicity, we ignore file names and the directory hierarchy again for the rest of this section. We restrict ourself to managing the contents of a single file. Instead of representing the contents of a file using an entry of the form $f = c$, we now use entries of the form $l = c$ to assign contents to a single line l. An entry of the form $l \rightarrow l'$ says that line l' follows line l. If the first line of f is l, we simply write $f \rightarrow l$. If f is empty, we write $f \rightarrow \mathsf{eof}$, using eof as a special line label to mark the end of a file.

We now describe how to insert a new empty line, how to modify the contents of a line, and how to delete an empty line:

$$\begin{aligned}&\mathit{insert}\ l\ l_b\ l_a = \{\, l_b \rightarrow l_a \,\} \vdash \{\, l = * \,\} \rightarrow \{\, l_b \rightarrow l, l \rightarrow l_a, l = \mathsf{empty} \,\} \\ &\mathit{modify}\ l\ c\ d = \{\, l = c \,\} \mapsto \{\, l = d \,\} \\ &\mathit{delete}\ l\ l_b\ l_a = \mathsf{inv}\ (\mathit{insert}\ l\ l_b\ l_a)\ .\end{aligned}$$

Note that an insertion or deletion records the labels of the surrounding lines, but not their contents. It is therefore possible to move a patch that inserts a line

into another repository even if in that repository the surrounding line has been modified. Of course, we should revisit our repository invariants. We refrain from doing so for the sake of brevity. This behaviour is, of course, once again only one point in the design space. It is easily possible to define all sorts of variations on the above patches.

Instead of further complicating the file system structure, we will now investigate some more abstract concepts often found in version control systems.

4.4 Logically connected changes

Often, changes to a repository only make sense in a group. The logic at the foundation of such a group is usually beyond a version control system to grasp and can only be specified by the user: all the changes might correspond to a new feature or bug-fix. While an individual patch might break the build, the set of patches as a whole work perfectly. We can model such a connection by allowing one patch to be defined as a composition of more basic patches $p_1, \ldots, p_n$. We denote such a *compound patch* simply as $(p_n \cdot \ldots \cdot p_1)$. A compound patch is treated as atomic: under no circumstances should any individual patch be applied without applying the entire compound patch.

4.5 Tags

A *tag* records the state of a repository at a certain time. It can be used as an easy way to identify a certain version of particular interest. In our model, we can tag the contents of a repository with internal representation X using the patch id_X. The tag does not change the representation, but merely earmarks a certain state.

4.6 Patch meta-data

A patch may store more information than just the changes in the file structure. Users may be interested in a patch's author, time of creation, or some form of documentation. We refer to all such additional information *meta-data*. Rather than fix the type of meta-data, we assume that there is a set M of all valid meta-data. We do not further specify its structure here, but leave that to the designers of a specific version control system.

There are several different ways to incorporate meta-data in our model. Meta-data can be part of the repository as an element of the form $\mathsf{meta}\ m$ with $m \in M$. Any patch can then be extended to produce a piece of meta-data using

$$\mathit{addmeta}\ p\ m = \mathsf{src}\ p \vdash \mathsf{ext}\ p \cup \{\mathsf{meta}\ m\} \rightarrow \mathsf{tgt}\ p \cup \{\mathsf{meta}\ m\}\ .$$

The function *addmeta* can be used on any patch, in particular also on compound patches as introduced earlier. A policy could then enforce that a repository should only contain patches that provide meta-data. Meta-data can also serve to distinguish otherwise identical patches.

4.7 Reverting changes

One of the main purposes of version control is the ability to return to a previous version by undoing a modification that later turns out to be undesired. For this, we make use of the fact that we require all patches to be invertible.

Note that we will distinguish undoing a patch p by deleting it from the repository, and undoing a patch p by adding $\mathsf{inv}\ p$ to the repository. The former reverts to a previous state and forgets everything about the change we remove, the latter also returns to the previous change, but keeps information about the patch in the repository. In the next section, we will formally define a repository as a multiset of patches, and then we can make this distinction precise.

4.8 Summary

We have seen how our theory of patches serves to express a multitude of concepts. Adding a new concept usually requires to adapt the entries we can have in our sets, and also to establish new invariants. We always strive to reflect the intent of patches as closely as possible in their representation. In particular, we have seen that our model is expressive enough to deal with line-based text files in a hierarchical directory structure. The same techniques, however, can also be applied to model other structures, such as files of structures formats that allow a different set of operations than the insertion and deletion of lines.

5 Communicating changes

So far we have defined what patches are and how they are created. The real fun starts once patches are communicated between repositories. Version control is not just about keeping track of a version history of documents, but also about communicating your changes to others.

5.1 Repositories

Before we can discuss communication between repositories, we need to define what a repository is.

Definition 8 (repository). A **repository** is a multiset of patches. The empty set defines the **empty repository** which we denote $\varnothing$.

There are a few interesting points to note about this definition. First of all, we allow the same patch to occur more than once. Restricting ourselves to a set would, for instance, disallow the multiset $\{p, \mathsf{inv}\ p, p\}$ where you change your mind about whether or not the patch p is desirable or not.

Our definition of repository does not record the order in which patches are performed. This may seem awkward – patches in a repository should have a natural, chronological order. An important observation, however, is that the order does not matter:

Definition 9 (consistent). A repository R is **consistent** if it can be written as $\{p_1, \ldots, p_n\}$ such that $(p_n \cdot \ldots \cdot p_1)$ is defined and applicable to the empty set. We call $X := (p_n \cdot \ldots \cdot p_1)\ (\varnothing)$ the **internal representation** of the repository.

Corollary 10. *The empty repository is consistent and has $\varnothing$ as its internal representation.*

Corollary 11. *The internal representation of a repository is well-defined, i.e. if π is a permutation of $\{1, \ldots, n\}$ and if $c_1{:=}(p_n \cdot \ldots \cdot p_1)$ and $c_2{:=}(p_{\pi\ (n)} \cdot \ldots \cdot p_{\pi\ (1)})$ are both defined and applicable to the empty set, then $c_1\ (\varnothing) = c_2\ (\varnothing)$.*

Proof. This follows directly from the commutativity of composition. □

In other words, whenever multiple orders of patches are valid, they all lead to the same result. This fact is important: we can consider repositories with the same patches to be equal, regardless of the order in which these patches were applied.

5.2 Moving patches

Interesting interactions occur when we want to move changes from one repository into another. Taking the perspective of the target repository, we say that we are *pulling* patches from another repository. Some version control systems place restrictions on which repositories may communicate. Such policies keep different repositories in step, but can be too restrictive. We shall describe the general case, leaving such restrictions to the discretion of specific systems.

All patches that are in the source repository, but not in ours, are eligible for pulling. If we are interested in a multiset of patches P, there are three options:

- moving P to our repository keeps our repository consistent,
- moving P creates an inconsistent repository, but we can pull a larger multiset P' in order to maintain consistency,
- all supersets of P that we could pull from the other repository would create an inconsistent repository.

The first two cases are covered by the following definition of a pull:

Definition 12 (pull). Given two consistent repositories R and S, a **pull** of patch multiset $P \subseteq R$ to S consists of a set $P' \subseteq (R \setminus S)$ such that $P \subseteq P'$ and $S \cup P'$ is a new consistent repository.

This definition of pull is still a bit too liberal: if P' is a pull, there may be many supersets of P' that are also pulls. This motivates the following definition:

Definition 13 (minimal pull). A pull M of some patch multiset P is said to be **minimal** if there is no pull N of P where $N \subset M$.

There may, however, be more than one minimal pull of a patch. In practice, a version control system should only pull in additional patches to resolve conflicts if there is a unique minimal pull. If there more than one minimal pull, the

user should be given the opportunity to further specify which pull should be performed.

The third case of above list – no successful pull involving the multiset we are interested in exists – is what we usually call a *conflict*. Conflicts must be resolved by inventing additional patches. Again, a version control system will usually involve the user in this process, and there is a lot of freedom in what to do.

Definition 14 (conflict resolution). If a patch multiset P causes a conflict when pulled to repository R, the **resolution** of P consists of a set of patches M such that $P \subseteq M$ and $R \cup M$ is consistent.

A resolution can either undo the pulled change completely, or undo some change in the target repository that was responsible for the conflict, or a combination of the two.

It is beyond the scope of this paper to discuss the algorithmic aspects or the user interface of communicating changes. It is relatively difficult to find minimal pulls or even determine conflicts in the normal case. Therefore, most actual version control systems impose certain restrictions on their repositories or use knowledge about the internal representations in order to perform these operations efficiently.

6 Related work and Discussion

Despite the large number of version control systems, most literature [2, 6, 7, 10, 13, 14] consists of manuals or technical documentation. There is surprisingly little work on version control in the scientific community. Existing work tends to focus on describing version control's place in the software development process [5] or describing specific instances of version control for structured documents [9]. When version control is mentioned in scientific literature, it is usually in the context of the much wider field of software configuration management.

The theory we present is most related to distributed version control systems, although classical centralized systems such as CVS [6] and Subversion [7] can of course be described in our theory. After all, a centralized repository layout is just a special case of a distributed layout where all client repositories are only allowed to communicate with the central repository.

Our work is similar in spirit to recent work on file synchronization [4, 11, 12]. Unison, for instance, is a widely used tool for synchronizing files between different machines. It exemplifies how a formal description of a problem can lead to the development of better tools. We hope that version control could benefit from a similar approach.

Darcs Darcs [8] is an advanced distributed version control system implemented in Haskell. Among the vast number of current distributed version control systems, darcs is special for two reasons.

First, it was the first (and apparently still is the only) real-life version control system that tried to focus on the *intent* of the user when it comes to patches. It keeps track of how information is moved within a file and how files are moved and renamed when merging changes. Because of the additional information available during merges, incorrect results and some conflicts are avoided. Darcs encourages so-called "cherry-picking", the selection of specific patches from another repository that are pulled without unnecessary dependencies. Note that the concept of intention-preservation is not new in general, but it is a first to see it put to use in a version control system.

Second, darcs makes an attempt to formalize the theory behind the system in a technical appendix, called the "theory of patches". Here, the semantics of darcs is described in a semi-formal way with references to quantum physics.

However, the theory is fairly sketchy. Some of the definitions are a bit vague, some proofs are missing, and only a few formal properties are stated. In fact, it was the vagueness of the theory of darcs that inspired our work: we wanted to develop a theory that is general enough to describe darcs, in particular, but can serve as a firm foundation for different flavours of version control systems.

In hindsight, it turns out that the fundamental difference between our system and the approach of darcs is that while darcs focuses on the intent of a patch, it leaves that intent implicit. Consider, for instance, line-based patches, represented in darcs using absolute lines offsets, whereas we use symbolic line labels. A line label does not change and uniquely identifies the logical line throughout its lifetime, whereas line offsets change when insertions or deletions are made at other places in the file. This leads to two disadvantages:

- it is difficult to say what exactly the intent of a line-based patch is, and
- the representation has to be adapted (i.e., the line numbers have to be recalculated) on several occasions.

A very practical disadvantage of having changing representations for one and the same patch is that patches cannot be signed, a field where darcs lags behind competitors such as monotone [3].

Conversely, in our theory the representation of patches is fixed. The freedom to choose a representation makes the intent of patches obvious. We give a framework that allows us to incorporate several patch types with several internal representations within the same theory. The number of patch types in darcs is relatively fixed and adapting it would require a significant amount of work, both as far as the theory is concerned as well as the implementation.

The advantages of darcs are, of course, on the practical side. The patches in darcs all carry meta-data so that they can effectively be compared. Repositories store patches in a certain order and cache a significant amount of information to avoid unnecessary recomputation. Dependencies between patches are easy to compute and stored explicitly. Because of dependencies, patches usually form a directed acyclic graph. In the case of a pull, it is therefore easy to see if a successful minimal pull containing the desired patches is possible or not.

We can express darcs's patches in our model. Although we have not discussed token replacement patches, they can be represented using the same techniques as

employed in Section 4. In turn, we hope that our theory could facilitate reasoning about darcs conflicts, a continuous hot topic in the darcs community.

Other distributed version control systems There are several other distributed version control systems that are currently in active use or development. While nearly all these systems record changes in individual patches, they have restrictions when it comes to cherry-picking, compared to darcs.

At the basis of the vast majority of systems is a variant of a three-way line-based merge algorithm that picks a common ancestor between two repositories. This algorithm may be augmented with simple informal extensions designed to handle file creations, deletions or renames, but fails to take further information about the intent of a patch into account. Therefore, pulling specific changes from other repositories can often lead to undesired results and requires a good understanding of the underlying algorithms. Codeville is noteworthy [1], as it makes use of an extended merge algorithm that applies certain heuristics to disambiguate where a three-way merge does not have enough information. However, while the algorithm may work well in practice, it is difficult to understand and hard to reason about.

In contrast, our theory allows a represention of text files in such a way that formally, merging is completely superfluous. When moving patches between repositories, the important question is not how to merge, but only whether the resulting repository is consistent. All the patches in a consistent repository that relate to one file are connected when interpreting the internal representation on an actual file system.

Several version control systems show amazing scalability and very good tool support for recording the changes made by the user on a disk, viewing the differences between repositories, and interactively resolving conflicts – areas that we have not focused on.

Conclusions and Further Work By presenting a generalized framework of version control, we have taken an important first step to enable reasoning about version control systems and develop an actual system that is founded on a firm theory. Nevertheless, there is still plenty of work to do.

While we have defined what constitutes a conflict, we have to investigate efficient algorithms for detecting and resolving conflicts. A naive algorithm for determining when a repository with n patches is inconsistent is $O\ (n!)$. In the future, we plan to investigate how to do better, and to identify special classes of repositories that allow more efficient operations.

We would also like to investigate the *algebraic* structure that patches and repositories carry. Corollary 5 and Definition 6 state that patches and internal representations form a *category*. *A fortiori*, because every patch is invertible we have a *groupoid*. Although we have presented the most basic algebraic properties of patches, their true nature is still terra incognita.

We believe that our approach has a great many merits. Our own experience with version control systems has been very much a love-hate relationship. We

could not live without them, but they are a source of continuous frustration. With the theory presented, version control might just have a brighter future ahead.

Acknowledgements We thank David Roundy and the darcs team for several inspiring discussions.

References

1. Codeville. `http://codeville.org`.
2. GNU arch. `http://www.gnu.org/software/gnu-arch/`.
3. monotone: distributed version control. `http://venge.net/monotone/`.
4. S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, October 1998.
5. Lars Bendix. *Configuration Management and Version Control Revisited*. PhD thesis, Aalborg University, December 1995.
6. Per Cederqvist. *Version Management with CVS*. Network Theory Ltd., 2002.
7. Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly, 2004.
8. David Roundy et al. Darcs: David's advanced revision control system. `http://www.darcs.net`.
9. David L. Hicks, John J. Leggett, and John L. Schnase. *Version Control in Hypertext Systems*. Texas A&M University, Hypermedia Research Lab, July 1991.
10. Mike Mason. *Pragmatic Version Control: Using Subversion*. Pragmatic Bookshelf, 2006.
11. Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.
12. Norman Ramsey and Elöd Csirmaz. An algebraic approach to file synchronization. In *Foundations of Software Engineering*, pages 175–185, 2001.
13. Garrett Rooney. *Practical Subversion*. Apress, 2006.
14. Jennifer Vesperman. *Essential CVS*. O'Reilly Media, 2003.

A Proof of Lemma 4

Because $g \cdot f$ is applicable to X, we know that

$$(E_1 \cup E_2) \cap X = S_1 \cup (S_2 \setminus T_1) \ .$$

If we intersect both sides of the equation with E_1, we get

$$(E_1 \cup E_2) \cap X \cap E_1 = (S_1 \cup (S_2 \setminus T_1)) \cap E_1 \ ,$$

which simplifies (using $E_1 \cap (S_2 \setminus T_1) = \varnothing$ and $S_1 \subseteq E_1$) to

$$E_1 \cap X = S_1 \ .$$

To show that g is applicable to $f\ (X)$, we start again from

$$(E_1 \cup E_2) \cap X = S_1 \cup (S_2 \setminus T_1) \ .$$

Now notice that the union on the right-hand side is disjoint, and therefore a symmetric difference. We apply $\triangle(T_1 \setminus S_2) \triangle T_1 \triangle S_1$ to both sides. Simplifying yields:

$$((E_1 \cup E_2) \cap X) \triangle (T_1 \setminus S_2) \triangle T_1 \triangle S_1) = S_2 \ .$$

Now we intersect both sides with E_2 and observe that $(T_1 \setminus S_2) \cap E_2 = \varnothing$ and $S_2 \subseteq E_2$:

$$(((E_1 \cup E_2) \cap X) \triangle T_1 \triangle S_1) \cap E_2 = S_2 \ .$$

Within an intersection with E_2, $((E_1 \cup E_2) \cap X) = X$, and therefore g is indeed applicable to $f\ (X)$.

The last part of the proof is surprisingly easy:

$$\begin{aligned}(g \cdot f)\ (X) &= (S_1 \cup (S_2 \setminus T_1)) \triangle X \triangle ((T_1 \setminus S_2) \cup T_2) \\ &= S_1 \triangle (S_2 \setminus T_1) \triangle X \triangle (T_1 \setminus S_2) \triangle T_2 \\ &= S_2 \triangle (S_1 \triangle X \triangle T_1) \triangle T_2 \\ &= g\ (f\ (X)) \ .\end{aligned}$$