Merge and Graft:

Two Twins That Need to Grow Apart

Robin La Fontaine

Nigel Whitaker

DELTAXML

Conference Paper: XML Prague

ABSTRACT

Software developers are familiar with merge, for example pulling together changes from one branch into another in a version control system. Graft (also known as ‘cherry pick’) is a more selective process, pulling changes from selected commits onto another branch. These two processes are often implemented in the same way, but they are not the same, there are subtle but important differences.

Git and Mercurial have different ways to determine the correct ancestor when performing a merge operation. Graft operations use yet another way to determine the ancestor. In the built-in line-based algorithm, the underlying diff3 process is then used. This diff3 algorithm accepts non-conflicting changes and relies on the user to resolve conflicts. We will examine the details of this process and suggest that the symmetrical treatment that diff3 uses is appropriate for merge but not necessarily optimal for the graft operation.

We will look at examples of tree-based structures such as XML and JSON to show how different approaches are appropriate for merge and graft, but the arguments presented apply to the merging of any structured content. Treating structured documents and data as tree-structured rather than just as a sequence of lines provides more intelligent merge and graft results, and enables rules to be applied to give improved results, thus reducing time-consuming and tedious manual conflict resolution.

We examine how these improvements can be integrated into version control systems, taking Git as an example, and suggest developments in their architecture that will make them even more powerful development tools.

CONTENTS

Introduction

Merging files across branches in a version control system can be a very time-consuming process. Automation is a great help, of course, but has its limitations, some of which are fundamental and some of which are due to the merge tools not fully understanding the source file structure.

One fundamental limitation is that it is not always possible to automate the propagation of changes because conflicts may occur. A conflict is a situation where accepting one change is not compatible with accepting another change – a choice needs to be made and so user intervention is needed. User intervention obviously slows down the process because it can take significant time to understand a conflict sufficiently to resolve it. A high level of expertise is needed to resolve conflicts and this is therefore an expensive process, not to mention being a very tedious job requiring sustained high levels of concentration.

It is worth mentioning that it is possible to resolve all conflicts automatically if a precedent order is defined, i.e. the changes in one branch will always take priority. We will discuss this in more detail later.

Merge tools are, usually, line-based tools that treat the files as a sequence of lines but they have no understanding of any syntactic structure that may be present. The result is often surprisingly good but can go very wrong in some situations, for example when the text has been re-formatted. Errors in the result can also occur because, for example, a new start tag insertion has been accepted but the corresponding end tag, perhaps several hundreds of lines further down the file, has been rejected – this type of error may be difficult to avoid, and very time consuming to correct.

Our concern in this paper is with processing XML, which has a tree structure, but the discussion also applies to other file types that can be converted into XML [1]. One example of this is JSON which also has a tree structure and can be converted into XML for processing [2] and then the result transformed back into JSON.

For XML, the merge process begins by merging all three source files into a single file where common data is shared and the differences between the three files are represented in a well-defined way. We call this intermediate merged file a delta file. Rules can then be applied to the delta file so that any changes are processed to generate a resultant merged file, ideally one that does not contain any conflicts. If there are conflicts, these can be represented in XML for downstream conflict resolution.

It is this ability to represent the three files in one, and the definition of rules to process the changes, that enables us to refine our understanding of the process and so identify subtle but important differences between the regular merge process and the graft process. This paper explores these differences as a step towards refining automated merge/graft processing to improve it and so save time and effort.

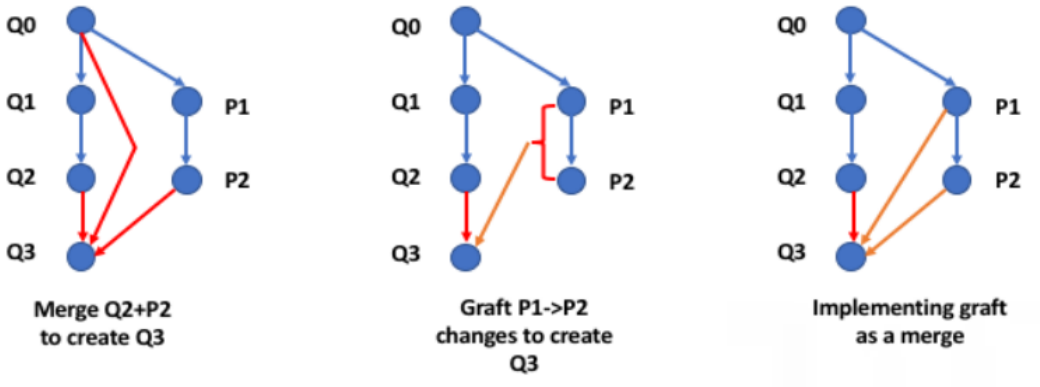
Terminology

First we will define our terminology and describe the merge and graft processes. Graft is also often known as ‘cherry pick’ but we will use the shorter term here.

Given two branches, each with two versions:

- Branch P with versions P0, P1 and P2
- Branch Q with versions Q0, Q1 and Q2

Figure 1. Merge and Graft Operations



The figure above shows the classic merge operation on the left. Graft, in the centre, cherry-picks the changes from P1 to P2 and applies this to Q2 to produce Q3. The figure on the right is the way that graft is implemented as a merge, so that P1 is considered the ancestor.

For merge, P0=Q0 in other words P1 and Q1 have a common ancestor.

The merge process is to create Q3 such that all the changes between Q0 and P2 and all the changes between Q0 and Q2 are represented in the changes between Q0 and Q3. This may result in conflicts that need to be resolved.

For graft, P0 may not equal Q0, but we want to propagate the changes between two selected commits on the same branch, such as P1 and P2, onto another branch, for example the one ending Q2, to create a new commit Q3.

These processes are not the same, and we will present XML and JSON examples to illustrate in detail how using the same algorithm for both does not always give optimal results.

The graft process is therefore to create Q3 such that all the relevant changes between P1 and P2 are represented in the changes between Q2 (the 'target' data set) and Q3. A change is only relevant if the data involved in the change is present in the target data set. This may also result in conflicts that need to be resolved, but typically there are fewer (or no) conflicts caused by graft than those caused by merge - for reasons that are described later.

A word about conflicts

As noted in the Introduction, it is possible to resolve all conflicts automatically if a precedent order is defined, i.e. the changes in one branch will always take priority. Conflicts are not usually resolved in this way because it does not always give the 'right' result simply because some conflicts require user review to understand why they have occurred and how best to correct them. The 'right' result may not be the acceptance of one change or the other but rather some form or merging of the changes. This is the case both for source code merge and for document merges.

There may be situations when a precedent order can be defined and if so time can be saved because some, if not all, conflicts can be resolved automatically. The ability to do this is a desirable feature in a

merge application.

How do merge and graft differ?

Regular three-way merge is a symmetrical process in that the two branches being merged are considered to have equal status. We do not want a change in one to override a change in the other without some user checking. Therefore we want conflicts to be identified. With tree structured data conflicts can be more complex, for example a sub-tree is deleted in one branch but it has also been changed in the other branch.

The use case for merge is well understood in version control systems, i.e. the requirement to merge changes from a branch back into the trunk or master version. The use case for graft is not so obvious, and it is worth considering use cases where graft is really what is required rather than merge. Within a version control system, graft is appropriate for 'cherry picking' changes between two versions on one branch to propagate these changes across to another branch. For source code, we might want to apply a bug fix but no other changes to another branch. It is also the appropriate process for keeping related sets of data synchronised in terms of changes made to either set.

Consider an example related to Master Data Management (MDM). We have a set of contact information (name, address, phone etc.) for members of an organisation. A smaller list or subset might be held for a local office, but how can this be kept up to date with changes to the master set of members? Regular three-way merge is not appropriate because a change to a member who is not in this office would result in an apparent conflict (amended in master set and deleted in local office set). Graft would work because this change would not be relevant so would be ignored. What about deletion? If a member leaves, then that member is deleted in the master set and both graft and merge would propagate this deletion to the local office set. But if a member moves from the local office to another office, then the deletion from the local office set would similarly propagate to the master set - which is not what is needed. This is an example where a 'safe graft' or 'graft without deletions' would be needed - a variation on the normal graft that requires different rules. Although this applies to a specific use case, this use case supports the argument that different use cases require different rules.

For regular merge, the appropriate common ancestor file is identified and each branch has the same relationship with it in that it has been derived from it. Therefore differences between the ancestor and each branch are 'real' changes that need to be considered. In some situations, the appropriate ancestor file is unambiguous, but in other more complex merge scenarios it may be necessary for more complex processing to determine what is appropriate. Further discussion of this is outside the scope of this paper.

For graft, on the other hand, we do not have a common ancestor. Therefore the graft situation is not symmetrical. We have two files from one branch and we are trying to apply the changes between these to some other file – typically a file that is very similar but there may be significant differences.

The principles can be illustrated with a simple example. This is a very simple example and seems rather trivial but the simplicity is intended to clarify the actual changes from the detail of the data.

Consider some file that contains contact information, identified by a name. To keep this very simple, we have used a string "v1" to represent the data rather than more realistic details of address, phone etc. Therefore "v2" represents some change to that data.

Given two branches, each with two versions:

- Branch P with versions P0, P1 and P2
- Branch Q with versions Q0, Q1 and Q2

We now consider and example to show that the implementation of graft as a merge does not always produce the desired results.

We started with P0 and Q0 containing these contacts: John, Mike, Anna. We will look at it as JSON, which is more compact than XML, but of course it could be XML.

Example 1. P0=Q0

```
{
  “John”: “v1”,
  “Mike”: “v1”,
  “Anna”: “v1”
}
```

We took these and added David into P1: John, Mike, Anna, David

Example 2. P1:

```
{
  “John”: “v1”,
  “Mike”: “v1”,
  “Anna”: “v1”,
  “David”: “v1”
}
```

We deleted John and changed details for Mike and David in P2: Mike has been changed and this is denoted by a new value “v2”, Anna is unchanged, and David is also changed, so again we indicate this simply with a new value “v2”

Example 3. P2:

```
{
  “Mike”: “v2”,
  “Anna”: “v1”,
  “David”: “v2”
}
```

The above describes what happened on the P branch. Now we can look at the changes made on the Q branch. First, we changed details for Anna in Q1: John, Mike, Anna is now “v2”.

Example 4. Q1:

```
{
  “John”: “v1”,
  “Mike”: “v1”,
  “Anna”: “v2”
```

}

We add a new contact, Jane, and change John in Q2: John is now “v2”, Mike, Anna is now “v2”, Jane

Example 5. Q2:

```
{
   “John”: “v2”,
   “Mike”: “v1”,
   “Anna”: “v2”,
   “Jane”: “v1”
}
```

A regular three-way merge of these two branches would result in a merged Q branch:

Q3 would have a conflict for John who has been deleted in P but changed in Q, Mike has value “v2”, Anna likewise is “v2”, Jane is unchanged and there is a conflict for David who appears to have been deleted in Q2 but at the same time has been changed in P.

Example 6. 3-way merge

P1	P2	Q2	Q3: 3-way merge

{
“John”: “v1”,
“Mike”: “v1”,
“Anna”: “v1”,
“David”: “v1”
}
	{
	“Mike”: “v2”,
	“Anna”: “v1”,
	“David”: “v2”
	}
		{
		“John”: “v2”,
		“Mike”: “v1”,
		“Anna”: “v2”,
		“Jane”: “v1”
		}
			{
			! CONFLICT
			“Mike”: “v2”,
			“Anna”: “v2”,
			! CONFLICT
			“Jane”: “v1”
			}

On the other hand, if we want to perform a graft then we want to propagate the changes between P1 and P2 onto the Q branch. The changes between P1 and P2 are to Mike and David, and John is deleted. However, as David does not appear in Q2 we cannot apply that change – it is not a conflict, it is just a change that is not relevant.

So the result is Q3: Mike has value “v2”, Anna has value “v2”, Jane is unchanged.

Example 7. Graft

P1	P2	Q2	Q3: Graft

{
“John”: “v1”,
“Mike”: “v1”,
“Anna”: “v1”,
“David”: “v1”
}
	{
	“Mike”: “v2”,
	“Anna”: “v1”,
	“David”: “v2”
	}
		{
		“John”: “v2”,
		“Mike”: “v1”,
		“Anna”: “v2”,
		“Jane”: “v1”
		}
			{
			“Mike”: “v2”,
			“Anna”: “v2”,
			“Jane”: “v1”
			}

Can we get the same result using a regular three way merge with a precedent on changes, e.g. that changes in Q override changes in P? In this case the deletion of David overrides the change in David in

the P branch. However, the change in John in Q overrides the deletion in P, so we would get:

Example 8. 3-way merge, Q priority

P1	P2	Q2	Q3: 3-way merge, Q priority
{ "John": "v1", "Mike": "v1", "Anna": "v1", "David": "v1" }	{ "Mike": "v2", "Anna": "v1", "David": "v2" }	{ "John": "v2", "Mike": "v1", "Anna": "v2", "Jane": "v1" }	{ "John": "v2", "Mike": "v2", "Anna": "v2", "Jane": "v1" }

Similarly, a merge with changes in P overriding changes in Q would leave David in the result which differs from the graft result, as shown below.

Example 9. 3-way merge, P priority

P1	P2	Q2	Q3: 3-way merge, P priority
{ "John": "v1", "Mike": "v1", "Anna": "v1", "David": "v1" }	{ "Mike": "v2", "Anna": "v1", "David": "v2" }	{ "John": "v2", "Mike": "v1", "Anna": "v2", "Jane": "v1" }	{ "Mike": "v2", "Anna": "v2", "David": "v2" "Jane": "v1" }

This shows that the rules for merge are different from the rules for graft. They may appear to be only different in a subtle way but the differences in the result of a complex merge are significant. So too would be the time saved in getting the automated result to be correct.

Some readers may say that the result of the graft is not exactly what they want because they do not want to ever have any data deleted, they only want additions to be propagated. Such a requirement might arise, for example, when we do not want master data deleted or changed. This is shown below.

Example 10. Graft: Additions only

P1	P2	Q2	Q3: Graft: Additions only
{ "John": "v1", "Mike": "v1", "Anna": "v1", "David": "v1" }	{ "Mike": "v2", "Anna": "v1", "David": "v2" }	{ "John": "v2", "Mike": "v1", "Anna": "v2", "Jane": "v1" }	{ "John": "v2", "Mike": "v1", "Anna": "v2", "Jane": "v1" }

The conclusion is that the merge or graft rules need to be controlled in different situations, and indeed may need to be controlled in a different way across an XML tree. The ability to define the rules according

to the use case is an advantage even if the standard definitions work in most situations.

We summarise the above results in the following table.

Table 1. Changes and their different interpretation for Merge and Graft

P1	P2	Q2	Merge interpretation	Graft interpretation	Comment
“John”: “v1”		“John”: “v2”	Deleted by P2, changed by Q2 - conflict	Deleted	We do not know if Q2 has changed this because we do not have its ancestor
“Mike”: “v1”	“Mike”: “v2”	“Mike”: “v1”	Changed by P2	Changed	
“Anna”: “v1”	“Anna”: “v1”	“Anna”: “v2”	Changed by Q2	No change to apply	
“David”: “v1”	“David”: “v2”		Changed by P2, deleted by Q2 - conflict	Data not present in target so cannot apply change	
		“Jane”: “v1”	Added by Q2	No relevant changes to apply	
	“Martin“=”v1”		Added by P2	Added	

The table above shows how the same situation is interpreted in a different way by merge and graft. Similarly, we can create a table to show a number of possible types of graft operation.

Table 2. Variants of Graft

P1	P2	Q2	‘Regular’ graft interpretation	‘Additions only’ graft interpretation	‘No deletions’ graft interpretation
“John”: “v1”		“John”: “v2”	Deleted	No change	No change
“Mike”: “v1”	“Mike”: “v2”	“Mike”: “v1”	Changed	No change	Changed
“Anna”: “v1”	“Anna”: “v1”	“Anna”: “v2”	No change to apply	No change	No change
“David”: “v1”	“David”: “v2”		Data not present in target so cannot apply change	No change	No change
		“Jane”: “v1”	No relevant changes to apply	No change	No change

P1	P2	Q2	'Regular' graft interpretation	'Additions only' graft interpretation	'No deletions' graft interpretation
	`"Martin"="v1"`		Added	Added	Added

These different flavours of graft are appropriate in different situations. For example if the Q branch is the master data then we might not want deletions to be applied. On the other hand if the P branch is the master one then we may want to apply all changes and deletions. In tree-structured data the situation is more complex because we may want to apply different rules at different places in the tree.

Advantages of XML for rule-based conflict resolution

Merge or graft rules can potentially become very adept at handling different situations, reducing the need to resolve conflicts by inspection.

It is very difficult to apply useful rules to line-based merges. There are two reasons for this. First, most data is structured in some way, for example the tree structure of XML and JSON. Source code is also structured and can typically be represented as a tree structure. Any line-based representation will not reflect this structure and so it is not possible to represent sensible rules. Secondly, the standard representations of change to text files, for example the output from diff or the diff3 format, do not easily support rule-based processing.

XML does provide significant advantages in this area. First, it is possible to merge several documents into one and represent the similarities and differences in XML. Given this representation, rules can then be defined to process the merged document to generate a result, so different sets of rules will generate different results according to the requirements. A major benefit of XML or JSON markup is that the conflicts can be *addressed* using mechanisms such as XPath [10] or JSON Pointer [9]. This enables much more powerful automated or rule-based processing. XPath and XSLT provide a well-defined language to specify the rules and execute them.

For example, consider the merge of a Maven POM file (a system for software build configuration). In the POM file we describe dependencies of the code being built and in a number of circumstances we have found version conflicts in POM dependencies. We could identify these conflicts with an `XPath: / project/ dependencies/dependency/version` and could say that for these conflicts we will use the version in the current or HEAD branch, or perhaps, assuming the dependency has an upwardly compatible API, we take the highest version.

It is the addressability of tree-based conflicts that introduces these automation possibilities. If conflict resolution rules can be applied in the merge driver then fewer conflicts would need to be presented to the user to deal with manually in a merge tool. We could expect that rules could be created to improve the conflict processing for a number of formats which are manually or semi-automatically created using developer tools such as IDEs, including Apache Ant, Maven POMs and XSLT.

Looking further ahead, other structured data could be converted into XML, and then back again, as shown with 'Invisible XML' 1. This approach opens up the opportunity to apply the intelligent, tree-structured XML merge to other types of data, then after application of rules the result can be converted back from XML into the original format. This is how the above JSON was processed 3.

Integration with Git Merge

We have argued here that 'merge' is not a fixed process, rather it changes according to different needs and the nature of the data. We now move on to discuss how we might go about implementing this for Git.

The merge (and also graft) process of Git involves a number of components, these are:

- merge scenario
- merge driver
- merge tool

The 'merge scenario' is responsible for looking at all of the files and directories with an understanding of moves and renames, matching up the corresponding files, determining the appropriate ancestor and calling the merge driver on triples of files. In some cases the scenario will determine that a full merge is unnecessary and may, for example, perform a fast-forward merge. It is also possible to specify a scenario such as 'mine' that produces a result that takes all of the files on a certain branch. In these cases it is not really a full merge and the merge driver may not be invoked.

The 'merge driver' receives three files corresponding to the ancestor and the two branches, loads the content into memory and is responsible for aligning their content and identifying any conflicts. Using a return code, it signals to the invoking code whether there are any conflicts. This usually reports a message to the user, often using a line starting with a capital "C" character. The diff3 merge driver in Git represents conflicts using a textual line based format consisting of marker lines using angle-bracket, equals or plus characters. The **git config** command can be used to configure different merge drivers and the `.gitattributes` file can then select between them using the filenames or extensions of the files being merged.

The user typically needs to resolve any conflicts in each file before the merge operation can be completed and committed. It is possible to take the file with the markers produced by the driver and resolve the conflicts by editing that output in a text editor and then reporting that the file has been resolved. A 'merge tool' provides a graphical user interface to automate the conflict resolution process, often allowing the user to select content from one of the branches or possibly the ancestor for each of the conflicting regions in the file.

There are two common usage or interaction patterns we have found relating to the use of merge drivers and merge tools:

- The merge driver produces the line-based conflict markers and then the merge tool reads the result file from the driver, interprets the markers and provides the user with selection capabilities based on this interpretation. Merge tools which take this approach include MS Visual Studio Code [5] and TkDiff [4].
- The merge tool, when it is invoked, is supplied with the filename of the driver result, but also the names of the original inputs to the merge driver. It can then re-merge the inputs and perhaps base its user interface on internal data structures from its own merge algorithm. Examples of tools which re-merge and do not seem to use the driver results include: Araxis Merge [7], P4Merge [6], and OxygenXML [8].

Given an enhanced, tree-based, three-way merge algorithm it is possible to integrate it into the Git merge process as either a merge driver or merge tool. We believe that the best approach is to integrate as a merge driver for reasons that we will summarise next.

Avoiding conflict confusion

The merge driver and merge tool should identify the same conflicts, i.e. behave in a consistent way. When processing XML with a line-based algorithm (such as diff3), changes such as those to attribute order might cause a conflict in the merge driver. In many workflows a conflicting file would cause the merge tool to be invoked in order to resolve the conflict. But if the merge tool then uses a tree- based XML or JSON aware algorithm this would not identify these apparent conflicts and the file may not even have any conflicts present. The unnecessary invocation of the merge tool may cause confusion for the user.

Improved non-conflicting results

A tree-based merge algorithm which is XML or JSON aware would normally produce well-formed XML or JSON results. However this is not true of a line based merge such as diff3, where the result may have mismatched element tags for example. These bad results will not necessarily be associated with a conflict - the mismatched tag may be non-conflicting. If the tree-aware algorithm is only used in the merge tool, it may never be invoked unless there is a conflict and it is there- fore possible for bad results to go unnoticed.

An algorithm with a better understanding of the data and its semantics can make better alignment decisions. Again in non-conflicting situations it makes sense to have this better alignment performed at the merge driver stage.

Simpler software design

The separation of the merge algorithm and a conflict resolving GUI can lead to simpler software design. It may be that merge tools find the textual markers insufficient for their needs and can provide a better experience by re-running a merge algorithm, but the merge architecture would be simpler if this was not necessary. This would avoid duplicated code and reduce the processing and IO required. In the following section we will look at this issue in greater detail.

Representing merge conflicts

It is possible to have an XML or JSON aware merge algorithm and then describe conflicts with diff3 compatible line-based markers discussed earlier. However, we found some limitations to this approach:

- If fine-grained textual changes are shown to the user on individual lines this can interfere with the original code or data layout and the result of the merge may be less familiar and need reformatting. Figure 2 shows one of our experiments in this area using Visual Studio code. It illustrates how an attribute con- flict can be reformatted to work well with the diff3 markers and Figure 3 shows the resolved result.
- The line based conflict format represents a linear sequence of conflicts. However tree based merge algorithms can have nested changes and (with an n-way or octopus merge) nested conflicts. Consider the case where one branch modifies a single word in a paragraph and in the other merge branch that paragraph is deleted. If the deletion is accepted then there is no need to descend further into that subtree to deal with any nested changes or conflicts.
- For XML, if a conflicting insertion of a start tag is accepted, then the corresponding end tag should

also be included, but there is no way to specify these linked edits in diff3. Similarly for JSON, it is not simple to ensure that the separating commas are included correctly.

Figure 2. XML attribute conflict in Microsoft Visual Studio Code

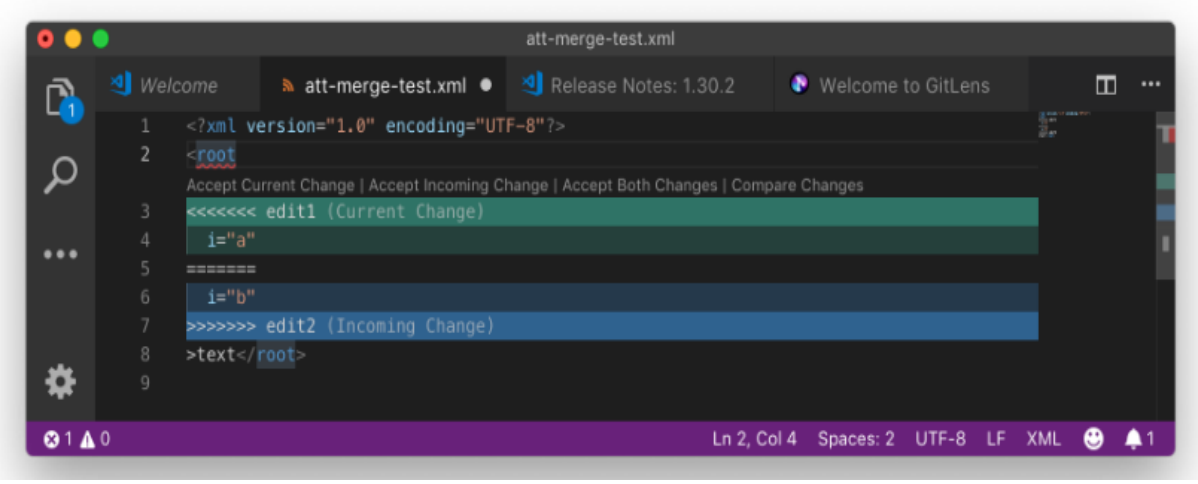
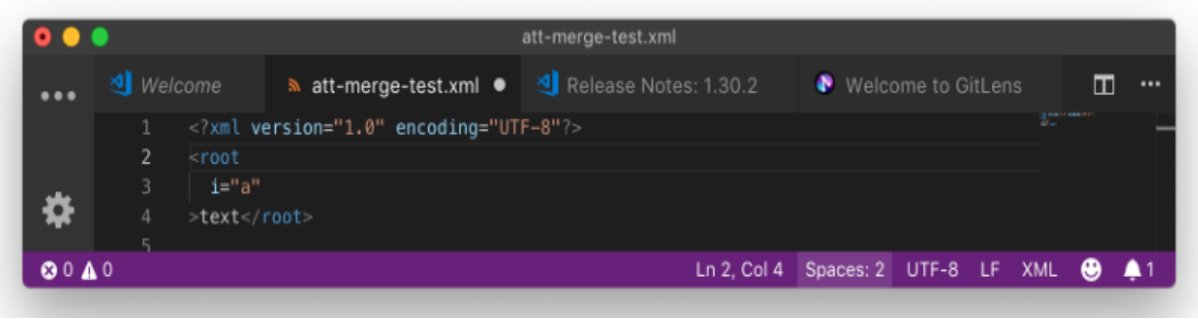


Figure 3. Resolved attribute conflict in Microsoft Visual Studio Code



These limitations, together with the limited number of merge tools which handle the diff3 format, have made us consider alternative approaches. Representing the conflict as XML or JSON markup allows us to overcome the limitations above and could allow better communication between the merge driver and merge tool. The intention would be to provide a rich enough change representation so that there is no need to re-run the merge algorithm. Merge tool applications can then take advantage of different merge drivers, and users can choose their preferred merge tool.

Some merge tools handle the data as lines of text, others are more aware of the type of data, e.g. OxygenXML understands the structure of XML or JSON. This leads to different requirements for the way that conflicts are communicated from the merge driver to the merge tool. A fully XML aware merge driver could communicate detailed XML semantics to an XML aware merge tool, but could only convey more limited information to a merge tool that understands only line based text files.

Conflict markers from diff3 may have line numbers but these are not helpful in free-format XML or JSON structures. On the other hand, users do care about layout for source code and for some JSON or XML data and this should be preserved. There is no simple solution to how best to communicate between more semantically intelligent merge drivers and the requirements and capabilities of various merge tools.

Conclusions

We have shown how merge and graft differ and argued that a single merge process is not appropriate to

cover both scenarios. This distinction can be taken further to show that there are not just two but many different types of merge, each useful and appropriate for different use cases.

The ability to represent structured data and documents as tree-based enables us to provide improved merge and graft operations. The use of XML, with XPath and XSLT, provides essential support to the development of a rule-based system that provides merge and graft for different use cases.

We have examined how such improvements can be integrated into Git and suggested developments in its architecture that will enable more versatile and appropriate merge and graft operations. The saving in time when dealing with large, complex documents and data is self-evident, but the advantage to users of reducing the time that needs to be spent on the very tedious task of conflict resolution is an even greater incentive to pursue this.

References

[1] Data Just Wants to Be Format-Neutral1, Steven Pemberton, CWI

[2] Transforming JSON using XSLT 3.02 Michael Kay, Saxonica

[3] JSON Compare Online Preview - DeltaXML https://www.deltaxml.com/json-client/

[4] tkdiff project3

[5] Visual Studio Code4

[6] Helix P4Merge and Diff Tool5

[7] Araxis Merge6

[8] Oxygen XML Editor7

[9] RFC: 6901 JavaScript Object Notation (JSON) Pointer8, Internet Engineering Task Force (IETF)

[10] XML Path Language (XPath) 3.19, W3C

Head Office (Sales and Support)

DeltaXML Ltd
Malvern Hills Science Park
Malvern, Worcestershire
WR14 3SZ UK

W: www.deltaxml.com
E: info@deltaxml.com
T: +44 1684 532 130

DELTAXML

[1] http://www.xmlprague.cz/day2-2016/#data
[2] http://www.xmlprague.cz/day2-2016/#jsonxslt
[3] https://sourceforge.net/projects/tkdiff/
[4] https://code.visualstudio.com
[5] https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge
[6] https://www.araxis.com/merge/
[7] https://www.oxygenxml.com/xml_editor.html
[8] https://tools.ietf.org/html/rfc6901
[9] https://www.w3.org/TR/2017/REC-xpath-31-20170321/