1 Introduction

I have started to look into the best way to use git based packaging methodology for Debian. As David Bremner says in his blog:

> The most natural way to work on a packaging project in version control is to have an upstream branch which either tracks upstream `Git/Hg/Svn`, or imports of tarballs (or some combination thereof, and a Debian branch where both modifications to upstream source and commits to stuff in `./debian` are added.

The challenge comes in translating the independent lines of development in `Git` branches to a serialized set of patches to the upstream source that are required for the *quilt (3.0)* source package format.

> The most obvious (and the most common) way to bridge the gap between `git` and `quilt` is to export patches manually (or using a helper like `gbp-pq`) and commit them to the packaging repository. This has the advantage of not forcing anyone to use `git` or specialized helpers to collaborate on the package.
>
> The next level of sophistication is to maintain a branch of upstream-modifying commits. Roughly speaking, this is the approach taken by `git-dpm`, by `gitpkg`, and with some additional friction from manually importing and exporting the patches, by `gbp-pq`. There are some issues with rebasing a branch of patches, mainly it seems to rely on one person at a time working on the patch branch, and it forces the use of specialized tools or workflows. Nonetheless, both `git-dpm` and `gitpkg` support this mode of working reasonably well.

At this point, David introduces `git-debcherry`, which uses an alternate method to serializing the various lines of development into a series of patches. This tool meets the following requirements.

- The user supplies two refs *upstream* and *head*. *upstream* should be suitable for export as a `.orig.tar.gz` file, and it should be an ancestor of *head*.
- At source package build time, we want to construct a series of patches that
  1. Is guaranteed to apply to *upstream*
  2. Produces the same work tree as *head*, outside `./debian`
  3. Does not touch `./debian`
  4. As much as possible, matches the git history from *upstream* to *head*.

At the project page git-dpm, there is a very nice step by step explanation of how one uses `git-dpm`. In this document is attempts a comparison of the effort and complexity when using `git-dpm` or `git-debcherry` to do equivalent packaging related tasks. To do that justice, a number of fairly routine tasks encountered in will be considered. To this end, consider the following scenario.

- An new package is created from an upstream source
- Next, a commit is made to work on feature `A` on top of the upstream code
- Next, and commit is made to start development of feature `B`
- Next, there is a new upstream version
- Throw in a change made to the `./debian` directory
- There follows an additional commit on the feature `B` branch
- A commit on the feature `A` branch

All in all, there are two upstream commit, two commits for feature `A`, and two commits for feature `B`, and one commit changing just the `./debian` directory. In the rest of this article, we will follow an hypothetical package through seven uploads.

2 Initial packaging

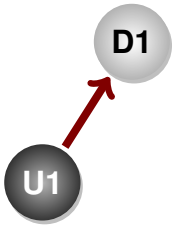

Figure 1: Initial packaging

The first step is common to both methods; we just import an upstream tarball and create an initial package. In Figure 1, each ball is associated with a commit, and the arrows represent a parent–child relationship between the commits.

```
git-import-orig U1.tar.gz
git checkout -b debian
hack ... git commit; git tag D1
```

3 Developing feature A: first commit

Now that we have an initial package, let us start improving it. So, we create a feature branch for the feature, and add a commit on that branch,

```
git checkout upstream, git co -b featureA
hack ... git commit ; git tag A1
```

3.1 Feature A1: Using debcherry

This is straightforward. Since the topic branch for feature A was just created, and this is the first commit on it, and the *debian* branch has had no other changes outside of `./debian`, we can just merge on to the *debian* branch.

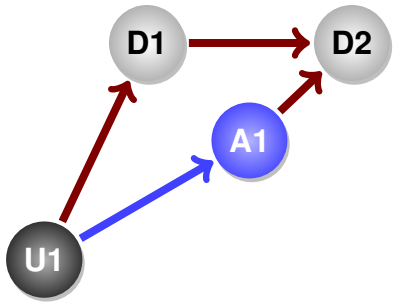

Figure 2: Adding a local feature: debcherry

```
git checkout debian, git merge A1                    (1)
hack ./debian ... git commit ; git tag D2            (2)
```

3.2 Feature A1: Using git-dpm

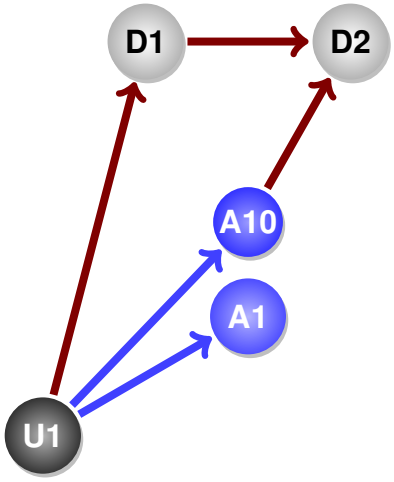

Figure 3: Adding a local feature: git-dpm

This where things get interesting. And somewhat complex. The history looks a bit busy.

```
git-dpm prepare                                      (1)
git-dpm checkout-patched                             (2)
git cherry-pick A1                                   (3)
git-dpm update-patches                               (4)
hack ./debian ... git commit ; git tag D2            (5)
```

4 Developing feature B: first commit

On to feature **B**

```
git checkout upstream, git co -b featureB
hack ... git commit ; git tag B1
```

4.1 Feature B1: Using debcherry

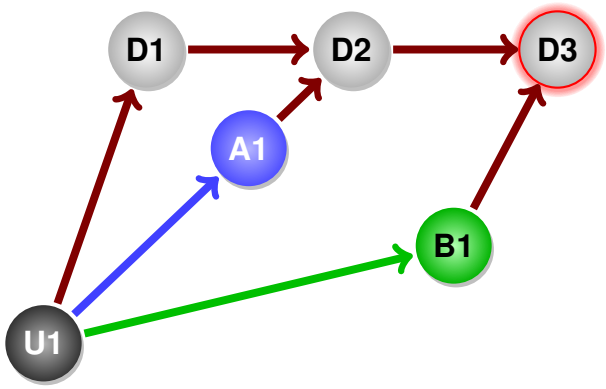

Figure 4: Starting a second feature: debcherry

This is still straightforward. We see our first use of the `git-debcherry` command. Please note that when we merge the new feature **B** commit on to the *debian* branch, there is a potential of needing to merge any conflicts that might happen.

```
git checkout debian, git merge B1                    (3)
git-debcherry -o debian/patches                      (4)
hack ./debian ... git commit ; git tag D3            (5)
```

4.2 Feature B1: Using git-dpm

Here the conflict resolution might need to happen when we cherry pick the commit for feature **B** on to the ephemeral *patched* branch. As far as conflict resolution steps go, either method has the same level of user pain. However, the history is beginning to look increasingly complex.

```
git-dpm checkout-patched                             (6)
git cherry-pick B1                                   (7)
git-dpm update-patches                               (8)
hack ./debian ... git commit ; git tag D3            (9)
```

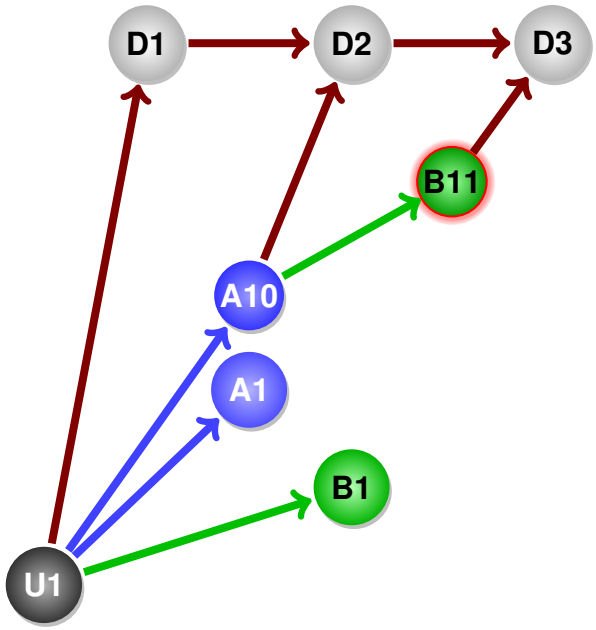

Figure 5: Starting a second feature: git-dpm

5 Changes to the Debian packaging and upstream

In this section we look at what happens if there is a change to the `./debian` directory, followed by a new upstream release. The following ignores any changes made in the new upstream that might require re-working the new features, since working through those scenarios adds more confusion than clarity at this point.

```
git checkout debian ... hack ... git commit ; git tag D4
git checkout upstream, git-import-orig U2.tar.gz; git tag U2
```

5.1 New upstream: using debcherry

I am beginning to like how one may just ignore the serializing issue to just before getting ready to package, build, test and upload. Most of the rest of the time one may just concentrate on normal development. Again, the point where the new upstream gets merged into Debian is a potential point where conflict resolution might be required.

```
git checkout debian, git merge U2                    (6)
git-debcherry -o debian/patches                      (7)
hack ./debian ... git commit ; git tag D5            (8)
```

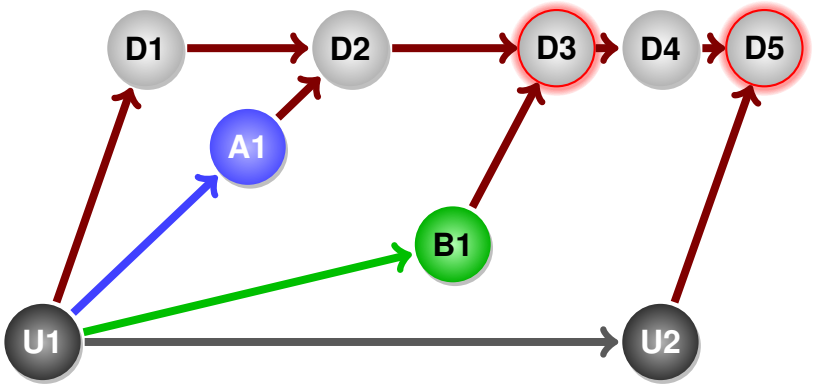

Figure 6: New upstream release: debcherry

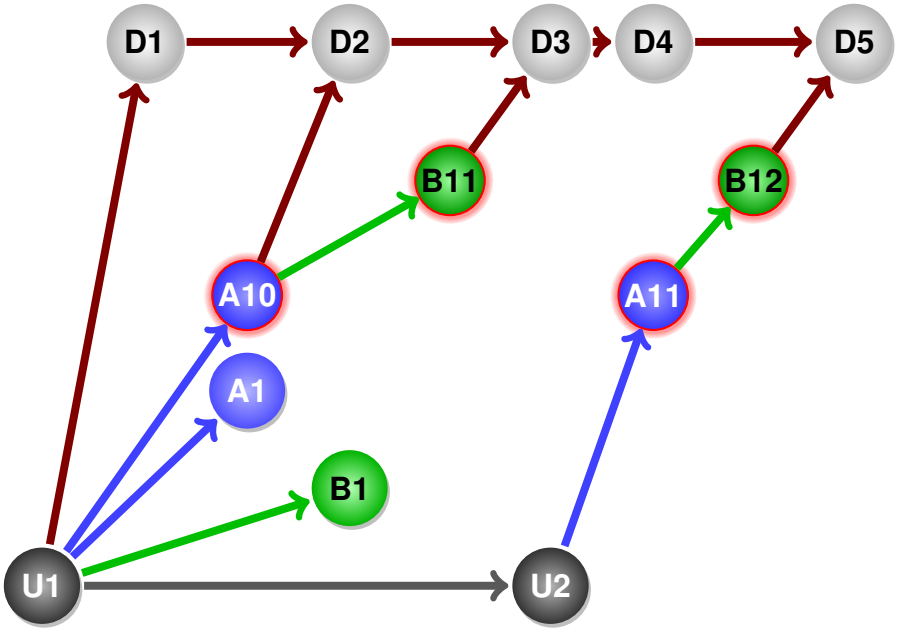

Figure 7: New upstream release: git-dpm

5.2 New upstream: Using git-dpm

```
git-dpm new-upstream                                    (10)
git-dpm rebase-patched                                  (11)
git-dpm update-patches                                  (12)
hack ./debian ... git commit ; git tag D5               (13)
```

Actually, `git-dpm new-upstream` could have been used to import the sources and rebase the *patched* branch on top of it with one go. Rebasing the previous patches might require conflict resolution. Looking at the two figures, we can see that using patch queue management using a patch branch is leading to a higher possibility of requiring conflict resolution, or, at least, more potential merge events where conflict resolution *might* be required.

6 Continuing development of feature B

The features are deliberately being updated in the reverse order of how they were initially started, just to see if it would cause any issues. There is little new in the way this commit is processed:

```
git checkout featureB
hack ... git commit ; git tag B2
```

6.1 Commit for feature B: using debcherry

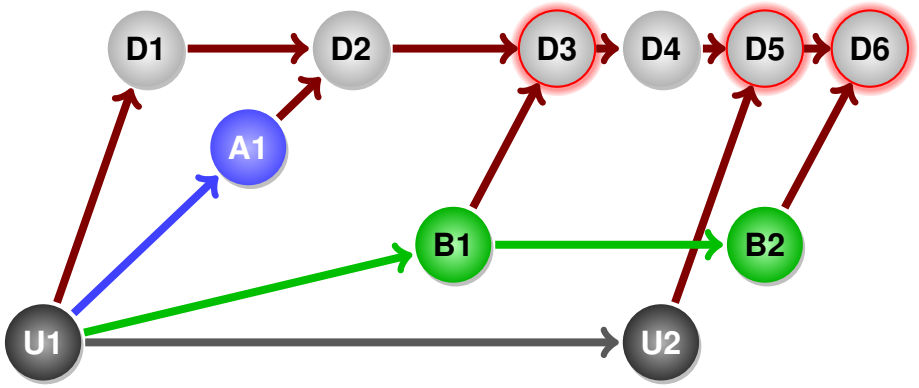

Figure 8: Enhancing feature B: debcherry

```
git checkout debian, git merge B2                       (9)
git-debcherry -o debian/patches                         (10)
hack ./debian ... git commit ; git tag D6               (11)
```

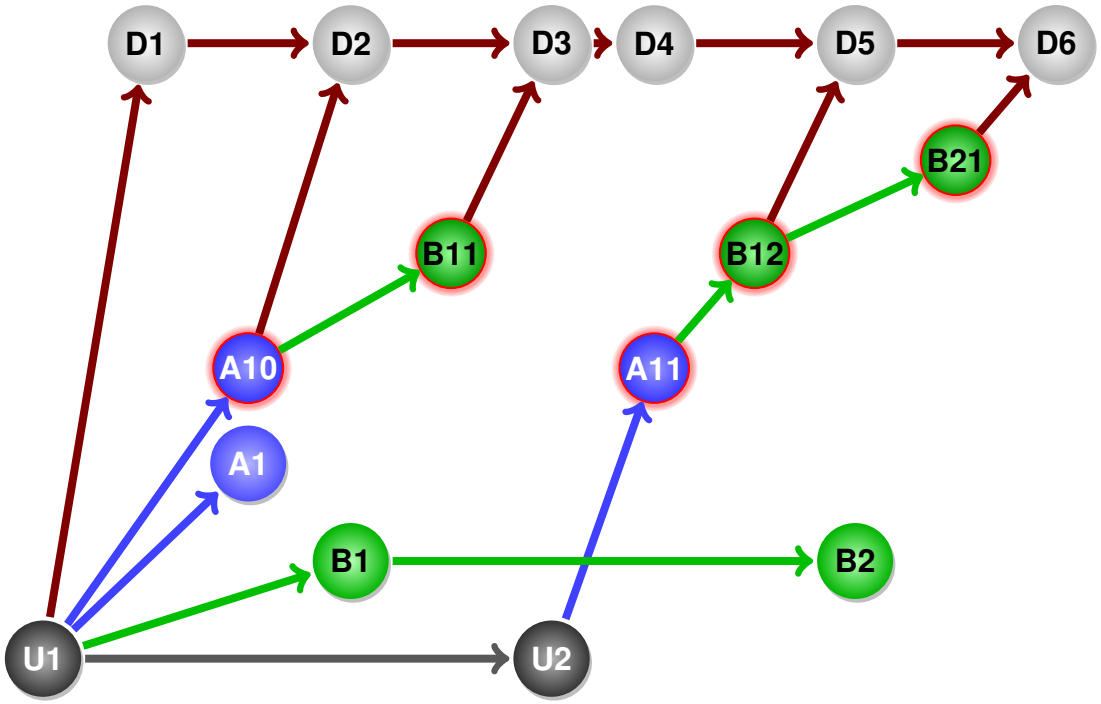

Figure 9: Enhancing feature B: git-dpm

6.2 Commit for feature B: Using git-dpm

While there is nothing new, please note that the history is beginning to get very complex here, compared to what we get when using `git-debcherry`. Such complexity has its own price. Also, though all the conflict resolution happens in the *patched* branches, the number of locations for possible conflict resolution does tend to climb.

```
git-dpm checkout-patched                                    (14)
git cherry-pick B2                                          (15)
git-dpm update-patches                                      (16)
hack ./debian ... git commit ; git tag D6                   (17)
```

7 Finally, updates to feature A

This is simple enough, and this is the last code drop in our exercise. We now have two commits each on the *upstream*. *feature A*, and *feature B* branches, as well as a pure packaging commit on the *debian* branch.

```
git checkout featureA
hack ... git commit ; git tag A2
```

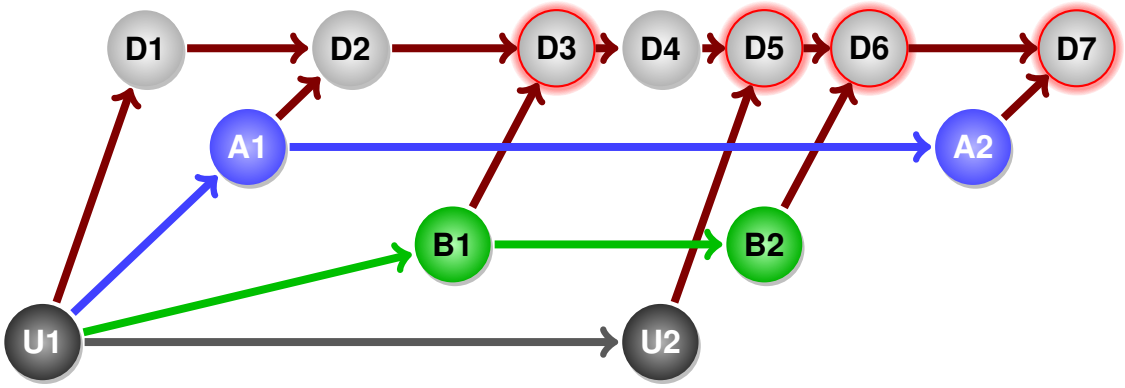

Figure 10: Enhancing feature A: debcherry

7.1 Commit for feature A: using debcherry

```
git checkout debian, git merge A2                    (12)
git-debcherry -o debian/patches                      (13)
hack ./debian ... git commit ; git tag D7            (14)
```

Using `git-debcherry` has been pretty simple, and the history is as minimal as it can be; there has been no impact on the history of the `git` repository due to the patch serialization required by the *quilt (3.0)* source format. Given this, this approach does seem to offer advantages over a serialization branch; although it is yet to be seen if approach will scale with a longer history of commits on the *debian* branch Also, all conflict resolution for merging all the lines of development into the packaged *debian* branch actually lives in the *debian* branch itself. That is neat.

7.2 Commit for feature A: Using git-dpm

By this time, the following routine should have become second nature::

```
git-dpm checkout-patched                             (18)
git cherry-pick A2                                   (19)
```

However, despite the fact that this is routine, the serialization process needing to be split up into two parts:

- Creating an ephemeral *patched* branch based on `./debian/patches`
- After working on the branch, recreating the patches from it using `git-dpm update-patches`

means that more activity takes place around patch serialization.

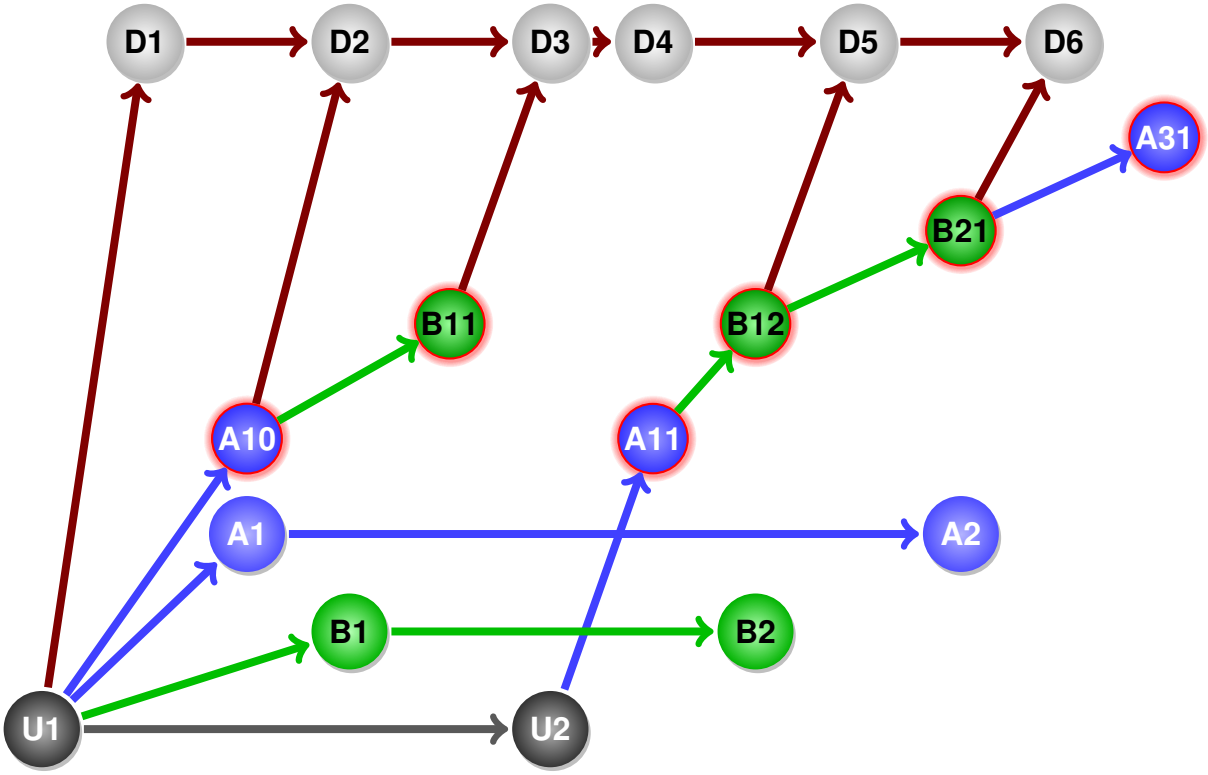

Figure 11: Enhancing feature A: git-dpm

At this point, we notice that our patched branch has the patches for feature **A** sandwiching the changes for feature **B**. We can improve on it by using `git rebase`

```
git rebase -i A11                                    (20)
git-dpm update-patches                               (21)
hack ./debian ... git commit ; git tag D6            (22)
```

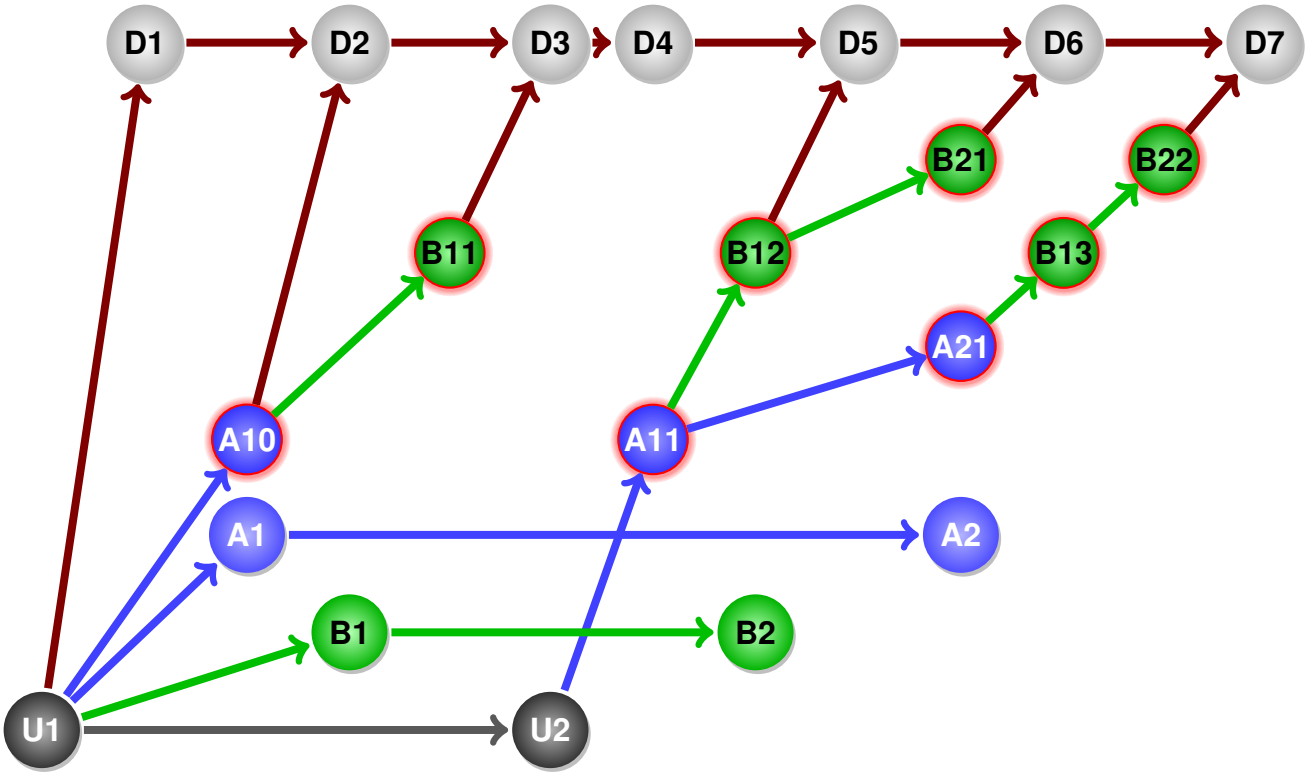

Figure 12: Enhancing feature A: git-dpm, part II

8 Conclusion

This concludes our exercise of following a hypothetical package through seven upload cycles, using two different strategies to serialize the lines of development. This has demonstrated that while `git debcherry` might be slower, and more complex, it has abstracted away some of the complexity of serializing `git` based development into the *quilt (3.0)* source format. It also tends to lead to a simpler history, which could be significant while debugging (or performing `git` bisects). However, there are some nice aspects of `git-dpm`, such as the ability to re-order patches by rebasing the *patched* branch, and not to lose any history by aggressively recording history for the ephemeral patch queue branch.