Inside a distributed version control system

Jim Hague
jim.hague@acm.org

May 2009

1 Preamble

Grinton Lodge is a Youth Hostel that sits on an exposed hillside just above the small hamlet of Grinton in Swaledale, in the Yorkshire Dales National Park. A former Victorian shooting lodge, it now welcomes walkers and other travellers from around the world.

Tonight, a Wednesday in mid-November, is not one of its busiest nights. Kat, the duty staff member, tells me that there is a small corporate team-building group in the annex. There's no sign of them at present. Otherwise, that portion of the world that has beaten a path to the door of this grand building today consists of just me. And Kat goes home soon.

The November CVu, removed from its wrappers and read yesterday, lies in my bag. Taunting me. Go on, it says, if you've ever going to put finger to keyboard in the name of CVu, well, tonight you are out of excuses.

Bugger.

2 Let's look into Mercurial

If you're at all interested in version control systems — and any software developer not using one daily is a strange beast indeed — you'll at least have become vaguely aware in the last few years of the growing maturity of the latest group of version control systems offering funky new stuff. These are the distributed version control systems (DVCS). There is more to them than just their headline attributes, being able to check history and do checkins while disconnected from a central server, but these are damn useful to start with.

When I first heard about DVCS, it wasn't immediately obvious to me (to put it mildly) how they would work. After years of using a centralised version control system, I had rough mental model of what went on. But how do you cope without the central server forcing ordering onto the changes?

Since then I've started using Mercurial[1]. Mercurial is a DVCS. It's one of three DVCSs

[1] http://www.selenic.com/mercurial

that have gained significant popularity in the last few years, the other two being Git[2] and Bazaar[3]. I switched a significant work project over to Mercurial (from Subversion) in mid-2007, because a customer site required on-site work but could not allow access back to the company VPN. I chose Mercurial for a variety of reasons which I won't bore you with here[4].

What I want to do in this article is give you an insight into how a DVCS works. OK, so specifically I'm going to be talking about Mercurial, but Git and Bazaar attack the problem in a similar way. But first I'd better give you some idea of how you use Mercurial.

2.1 The 5 minute Mercurial overview

2.1.1 The basics

I think it unlikely that someone possessing the taste and discernment to be reading CVu would not be familiar with at least one version control system. So, while I want to give you a flavour of what it's like to use, I'm not going to hang about. If you'd like a proper introduction, or you don't follow something, I thoroughly recommend you consult the Mercurial book.

To start using Mercurial to keep track of a project.

```
$ hg init
$
```

This creates the repository root in the current directory.

Like CVS[5] with its `CVS` directory and Subversion[6] with its `.svn` directory, Mercurial keeps its private data in a directory. Mercifully there is only one of these, in the top level of your project. And rather than holding details of where the actual repository is to be found, the `.hg` directory holds the entire repository.

[2] http://git-scm.com
[3] http://bazaar-vcs.org/
[4] OK, if you must know:

- Implementability. I needed the system to work on Windows, Linux and AIX. The latter was not one of the directly supported platforms for any of the candidates. Git's implementation uses a horde of tools. Bazaar requires only Python, but required Python 2.4 while IBM stubbornly still supplies only Python 2.3. Mercurial requires Python 2.3 or greater, and uses some C for speed.
- Simplicity. My users used Subversion daily, but did not generally have much experience with other VCS. From the command line, Mercurial's core operations will be familiar to a Subversion user. This is also true of Bazaar, but was less true of Git. Git has improved in this matter since then, but a Mr Winder of this parish tells me that it's still possible to seriously embarrass yourself. There was also a lack of Windows support for Git at the time.
- Speed. Mercurial is fast. In the same ballpark as Git. Bazaar wasn't, and although it has improved significantly, has, in my estimation, added user complexity in the process, and at the time of writing is still off the pace for some operations.
- Documentation. At the time, Bryan O'Sullivan's excellent Mercurial book (http://hgbook.red-bean.com) was a clear winner for best documentation.

[5] http://www.nongnu.org/cvs/
[6] http://subversion.tigris.org/

Next you need to specify the files you want Mercurial to track.

```
$ echo "There was a gibbon one morning" > pome.txt
$ hg add pome.txt
$
```

As you might expect, this marks the files as to be added. And as you might also expect, you need to commit to record the added files in the repository. The commit comment can be supplied on the command line; if you don't supply a comment, you'll be dropped into an editor to provide one.

There is a suggested format for these messages — a one line summary followed by any more required detail on following lines. By default Mercurial will only display the first line of commit messages when listing changes. In these examples I'll stick to terse messages, and I'll enter them from the command line.

```
$ hg commit -m "My Pome" -u "Jim Hague <jim.hague@acm.org>"
$
```

Mercurial records the user making the change as part of the change information. It is usual to give your name and email address as I've done here. You can imagine, though, that constantly having to repeat this is a bit tedious, so you can set a default user name in a configuration file. Mercurial keeps global, user and repository configurations, and it can go in any of those.

As with Subversion, after further edits you see how your working copy differs from the repository.

```
$ hg status
M pome.txt
$ hg diff
diff -r 33596ef855c1 pome.txt
--- a/pome.txt  Wed Apr 23 22:36:33 2008 +0100
+++ b/pome.txt  Wed Apr 23 22:48:01 2008 +0100
@@ -1,1 +1,2 @@ There was a gibbon one morning
 There was a gibbon one morning
+said "I think I will fly to the moon".
$ hg commit -m "A great second line"
$
```

And look through a log of changes.

```
$ hg log
changeset:   1:3d65e7a57890
tag:         tip
user:        Jim Hague <jim.hague@acm.org>
date:        Wed Apr 23 22:49:10 2008 +0100
summary:     A great second line
```

```
changeset:   0:33596ef855c1
user:        Jim Hague <jim.hague@acm.org>
date:        Wed Apr 23 22:36:33 2008 +0100
summary:     My Pome

$
```

There are some items here that need an explanation.

The changeset identifier is in fact two identifiers separated by a colon. The first is the sequence number of the changeset in the repository, and is directly comparable to the change number in a Subversion repository. The second is a globally unique identifier for that change. As the change is copied from one repository to another (this is a distributed system, remember, even if we haven't come to that bit yet), its sequence number in any particular repository will change, but the global identifier will always remain the same.

`tip` is a Mercurial term. It means simply the most recent change.

Want to rename a file?

```
$ hg mv pome.txt poem.txt
$ hg status
A poem.txt
R pome.txt
$ hg commit -m "Rename my file"
$
```

(The command to rename a file is actually `hg rename`, but Mercurial saves Unix-trained fingers from typing embarrassment.)

At this point you may be wondering about directories. `hg mkdir` perhaps? Well, no. Mercurial only tracks files. To be sure, the directory a file occupies is tracked, but effectively only as a component of the file name. This has the slightly unexpected result that you can't record an empty directory in your repository.[7]

Given this, and the status output above that suggests strongly that Mercurial treats a rename as a copy followed by a delete, you may be worried that Mercurial won't cope at all well with rearranging your repository. Relax. Mercurial does store the details of the rename as part of the changeset, and copes very well with rearrangements[8].

Want to rewind the working copy to a previous revision?

```
$ hg update -r 1
1 files updated, 0 files merged, 1 files removed, 0 files unresolved
$
```

[7] I tripped over this converting a work Subversion repository. One possibility is to create a placeholder file in the directory. In the event I created the directory (which receives build products) as part of the build instead.

[8] The Mercurial designers justify not dealing with directories as first class objects by pointing out that provided you can correctly move files about in the tree, the other reasons for tracking directories are uncommon and do not in their opinion justify the considerable added complexity. So far I've found no reason to doubt that judgement.

`hg update` updates the working files. In this case I'm specifying that I want to go back to local changeset 1. I could also have typed `-r 3d65e7a57890`, or even `-r 3d`; when specifying the global change identifier you only need to type enough digits to make it unique.

This is all very well, but it's not exactly distributed, is it?

2.1.2 Going distributed

A version control system goes Distributed by allowing multiple copies of the repository to exist, and work to be done in all those repositories in parallel. So when you start work on an existing project, the first thing to do is to get your own copy of the repository.

```
elsewhere$ hg clone ssh://jim.home.net/Poem Jim-Poem
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Mercurial lets you access other repositories via the file system, over http or over ssh.

```
elsewhere$ cd Jim-Poem
elsewhere$  hg log
changeset:   3:a065eb26e6b9
tag:         tip
user:        Jim Hague <jim.hague@acm.org>
date:        Thu Apr 24 18:52:31 2008 +0100
summary:     Rename my file

changeset:   2:ff97668b7422
user:        Jim Hague <jim.hague@acm.org>
date:        Thu Apr 24 18:50:22 2008 +0100
summary:     Finished first verse

changeset:   1:3d65e7a57890
user:        Jim Hague <jim.hague@acm.org>
date:        Wed Apr 23 22:49:10 2008 +0100
summary:     A great second line

changeset:   0:33596ef855c1
user:        Jim Hague <jim.hague@acm.org>
date:        Wed Apr 23 22:36:33 2008 +0100
summary:     My Pome

$
```

`hg clone` is aptly named. It creates a new repository that contains exactly the same changes as the source repository. You can make a clone just by copying your project directory, if you're confident nothing else will access it during the copy. `hg clone` saves you this worry, and sets the default push/pull location in the new repo to the cloned repo.

From that point, you use `hg pull` to collect changes from other places into your repo (though note it does not by default update your working copy), and, as you might guess, `hg push` shoves your changes into a foreign repository. By default these will act on the repository you cloned from, but you can specify any other repository.

More on those in a moment. First, though, I want to show you something you can't do in Subversion. Start with the repository with 4 changes we just cloned. I want to focus on the first couple of lines, so I'll wind the working copy back to the point where only those lines exist.

```
$ hg update -r 1
1 files updated, 0 files merged, 1 files removed, 0 files unresolved
$
```

And make a change.

```
$ hg diff
diff -r 3d65e7a57890 pome.txt
--- a/pome.txt  Wed Apr 23 22:49:10 2008 +0100
+++ b/pome.txt  Thu Apr 24 19:13:14 2008 +0100
@@ -1,2 +1,2 @@ There was a gibbon one morning
-There was a gibbon one morning
-said "I think I will fly to the moon".
+There was a baboon who one afternoon
+said "I think I will fly to the sun".
$ hg commit -m "Better first two lines"
$
```

The alert among you will have sat up at that. Well done! Yes, there's something very worrying. How can I commit a change at an old point? If you try this in Subversion, it will complain mightily about your file being out of date. But Mercurial just went ahead and did something. The Bazaar experts among you will know that in Bazaar, if you use `bzr revert -r` to bring the working copy to a past revision, make a change and commit, then your latest version will be the past revision plus your change. Perhaps that's what Mercurial did?

No. What Mercurial did is central to Mercurial's view of the world. You took your working copy back to an old changeset, and then committed a fresh change based at that changeset. Mercurial actually did just what you asked it to do, no more and no less. Let's see the initial evidence.

```
$ hg heads
changeset:   4:267d32f158b3
tag:         tip
parent:      1:3d65e7a57890
user:        Jim Hague <jim.hague@acm.org>
date:        Thu Apr 24 19:13:59 2008 +0100
summary:     Better first two lines

changeset:   3:a065eb26e6b9
```

```
user:        Jim Hague <jim.hague@acm.org>
date:        Thu Apr 24 18:52:31 2008 +0100
summary:     Rename my file

$
```

Time for some more Mercurial terminology. You can think of a `head` in Mercurial as the most recent change on a branch. In Mercurial, a branch is simply what happens when you commit a change that has as its parent a change that already has a child. Mercurial has a standard extension `hg glog` which uses some ASCII art to show the current state:

```
$ hg glog
@  changeset:   4:267d32f158b3
|  tag:         tip
|  parent:      1:3d65e7a57890
|  user:        Jim Hague <jim.hague@acm.org>
|  date:        Thu Apr 24 19:13:59 2008 +0100
|  summary:     Better first two lines
|
| o  changeset:   3:a065eb26e6b9
| |  user:        Jim Hague <jim.hague@acm.org>
| |  date:        Thu Apr 24 18:52:31 2008 +0100
| |  summary:     Rename my file
| |
| o  changeset:   2:ff97668b7422
|/   user:        Jim Hague <jim.hague@acm.org>
|    date:        Thu Apr 24 18:50:22 2008 +0100
|    summary:     Finished first verse
|
o  changeset:   1:3d65e7a57890
|  user:        Jim Hague <jim.hague@acm.org>
|  date:        Wed Apr 23 22:49:10 2008 +0100
|  summary:     A great second line
|
o  changeset:   0:33596ef855c1
   user:        Jim Hague <jim.hague@acm.org>
   date:        Wed Apr 23 22:36:33 2008 +0100
   summary:     My Pome

$
```

`hg view` shows a nicer graphical view[9].

So the change is in there. It's the latest change, and is simply on a different branch to the other changes.

Almost invariably, you will want to bring everything back together and merge the branches. A merge is a change that combines two heads back into one. It prepares

[9]Though, being Tcl/Tk based, not that much nicer.

an updated working directory with the merged contents of the two heads for you to review and, if satisfactory, commit.

```
$ hg merge
merging pome.txt and poem.txt
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ cat poem.txt
There was a baboon who one afternoon
said "I think I will fly to the sun".
So with two great palms strapped to his arms,
he started his takeoff run.
$ hg commit -m "Merge first line branch"
$
```

(I'm no poet. The poem is, of course, *Silly Old Baboon* by the late, great, Spike Milligan. From *A Book of Milliganimals*, Puffin, 1971.)

Here's the ASCII art again showing what just happened. Oh, and notice in the above that Mercurial has done the right thing with regard to the rename.

```
$ hg glog
@    changeset:   5:792ab970fc80
|\   tag:         tip
| |  parent:      4:267d32f158b3
| |  parent:      3:a065eb26e6b9
| |  user:        Jim Hague <jim.hague@acm.org>
| |  date:        Thu Apr 24 19:29:53 2008 +0100
| |  summary:     Merge first line branch
| |
| o  changeset:   4:267d32f158b3
| |  parent:      1:3d65e7a57890
| |  user:        Jim Hague <jim.hague@acm.org>
| |  date:        Thu Apr 24 19:13:59 2008 +0100
| |  summary:     Better first two lines
| |
o |  changeset:   3:a065eb26e6b9
| |  user:        Jim Hague <jim.hague@acm.org>
| |  date:        Thu Apr 24 18:52:31 2008 +0100
| |  summary:     Rename my file
| |
o |  changeset:   2:ff97668b7422
|/   user:        Jim Hague <jim.hague@acm.org>
|    date:        Thu Apr 24 18:50:22 2008 +0100
|    summary:     Finished first verse
|
o  changeset:   1:3d65e7a57890
|  user:        Jim Hague <jim.hague@acm.org>
|  date:        Wed Apr 23 22:49:10 2008 +0100
|  summary:     A great second line
```

```
|
o  changeset:   0:33596ef855c1
   user:        Jim Hague <jim.hague@acm.org>
   date:        Wed Apr 23 22:36:33 2008 +0100
   summary:     My Pome

$
```

So, our little branch change has now been merged back, and we have a single line of development again. Notice that unlike the other changesets, changeset 5 has two parent changesets, indicating it is a merge changeset. You can only merge two branches in one operation; or putting it another way, a changeset can have a maximum of two parents.

This behaviour is absolutely central to Mercurial's philosophy. If a change is committed that takes as its starting point a change that already has a child, then a branch gets created. Working with Mercurial, branches get created frequently, and equally frequently merged back. As befits any frequent operation, both are easy to do.

You're probably thinking at this point that this making a commit onto an old version is a slightly strange thing to do, and you'd be right. But that's exactly what's going to happen the moment you go distributed. Two people working independently with their own repositories are going to make commits based, typically, on the latest changes they happen to have incorporated into their tree. To be Distributed, a DVCS has to deal with this. Mercurial faces it head-on. When you pull changes into your repo (or someone else pushes them), if any of the changes overlap — are both based on the same base change — you get extra heads, and it's up to you to let these extra heads live or merge, as you please.

In practice this is more manageable then you might think. Consider a typical Mercurial usage, where the 'master' repo sits on a known server, and everyone pulls changes from the master and pushes their own efforts to the master. By default Mercurial won't let you push if the receiving repo will gain an extra head as a result, so you typically pull (and do any required merging) just before pushing. Subversion users will recognise this pattern. Subversion won't let you commit a change if your working copy is not at the very latest revision, so the Subversion user will update, and merge if necessary, just before committing.

What, then, about a branch in the conventional sense of '1.0 maintenance branch'? Typically in Mercurial you'd handle this by keeping a separate cloned repository for those changes. Cloning is fast, and if local uses hard links where possible on filesystems that support them, so isn't necessarily extravagant on disc space. You can, if you prefer, handle them all in a single repo with 'named branches', but cloning is definitely simpler.

OK, so now you know the basics of using Mercurial. We can proceed to looking at how this magic is achieved. In particular, where does this magic globally unique identifier for a change come from?

2.2 Inside the Mercurial repo

The way Mercurial handles its repo is really quite simple.

That's simple, as in 'most things are simple once you know the answer'. I found the explanation helpful[10], so this section attempts the 10,000ft (FL100 if you prefer) view of Mercurial.

First remember that any file or component can only have one or two parents. You can't merge more than one other branch at once.

We start with the basic building block, which Mercurial calls a revlog. A revlog is a thing that holds a file and all the changes in the file history[11]. The revlog stores the differences between successive versions of the file, though it will periodically store a complete version of the file instead of a difference, so that the content of any particular file version can always be reconstructed without excessive effort.

Under the secret-squirrel Mercurial `.hg` directory at the top of your project is a store which holds a revlog for each file in your project. So you have the complete history of the project locally. No more round trips to the server.

Both the differences between successive versions and the periodic complete versions of a file are compressed before storing. This is surprisingly effective at minimising the storage requirements this entire history of your project. I have a small Java project handy, comprising a little over 300 source modules. There are 5 branches plus the mainline, and some 1920 commits in all. A Subversion checkout of the current mainline takes 51Mb. Converting the project to Mercurial yields a Mercurial repository that takes 60Mb, so a little bigger. Remember, though, that the Mercurial repository includes not just the working copy, but also the entire history of the project.

Any point in the evolution of a revlog can be uniquely identified with a nodeid. This is simply the SHA1 hash of the current file contents concatenated with the nodeids of one or both parents of the current revision. Note that this way, two file states are identical if and only if the file contents are the same *and* the file has the same history.

Here's a dump of a revlog index:

```
$ hg debugindex .hg/store/data/pome.txt.i
   rev    offset  length   base linkrev nodeid       p1           p2
     0         0      32      0       0 6bbbd5d6cc53 000000000000 000000000000
     1        32      51      0       1 83d266583303 6bbbd5d6cc53 000000000000
     2        83      84      0       2 14a54ec34bb6 83d266583303 000000000000
     3       167      76      3       4 dc4df776b38b 83d266583303 000000000000
$
```

Note here that a file state can have two parents. If both the parent nodeids are non-null, the file state has two parents, and the state is therefore the result of a merge.

Let's dump out a revlog at a particular revision:

```
$ hg debugdata .hg/store/data/pome.txt.i 2
There was a gibbon one morning
said "I think I will fly to the moon".
So with two great palms strapped to his arms,
```

[10]For the curious, Bryan O'Sullivan's excellent Mercurial book has a chapter on the subject, and the Mercurial website has a fair amount of detail too.

[11]For any non-trivial file, this will actually be two files on the disc, a data file and an index.

```
he started his takeoff run.
$
```

The next component is the manifest. This is simply a list of all the files in the project, together with their current nodeids. The manifest is a file, held in a revlog. The nodeid of the manifest, therefore, identifies the project filesystem at a particular point.

```
$ hg debugdata .hg/store/00manifest.i 5
poem.txt5168b1a5e2f44aa4e0f164e170820845183f50c8
$
```

Finally we have the changeset. This is the atomic collection of changes to a repository that leads to a new revision. The changeset info includes the nodeid of the corresponding manifest, the timestamp and committer ID, a list of changed files and a comment. The changeset also includes the nodeid of the parent changeset, or the two parents if the change is a merge. The changeset description is held in a revlog, the changelog.

```
$ hg debugdata .hg/store/00changelog.i 5
1ccc11b6f7308cc8fa1573c2f3811a4710c91e3e
Jim Hague <jim.hague@acm.org>
1209061793 -3600
poem.txt
pome.txt

Merge first line branch
$
```

The nodeid of the changeset, therefore, gives us a globally unique identifier for any particular change. Changesets have a Subversion-like incrementing change number, but it is peculiar to that repository. The nodeid, however, is global.

One more detail remains to complete the picture. How do we get back from a particular file change to find the responsible changeset? Each revlog change has a linkrev entry that does just this.

So, now we have a repository with a history of the changes applied to that repository. Each change has a unique identifier. If we find that change in another repository, it means that at the point in the other repository we have exactly the same state; the file contents and history are identical.

At this point we can see how pulling changes from another repository works. Mercurial has to determine which changesets in the source repository are missing in the target repository. To do this, for each head in the source repo it has to find the most recent change in that head that it already present in the target repo, and get any remaining changes after that point. These changes are then copied over and applied.

The Mercurial revlog format has proved remarkably durable. Since the first release of Mercurial in April 2005, there have been a total of 5 changes to the file format. However, of those, all but one have been changes to the handling of file names. The most recent change, in October 2008, and its predecessor in December 2006, were both introduced purely to cope with Windows specific issues. The one change that touched the data structures described above was in April 2006. The format introduced, RevLogNG, changed only the details of index data held, not the overall design. The chief Mercurial developer, Matt Mackall, notes that the code in present-day Mercurial devoted to reading the old format comprises 28 lines of Python. Compared with, say, the early tribulations of Subversion and the switch from `bdfs` to `fsfs`, this is an impressive record.

3 Reflections on going distributed

It's nearly traditional at this stage in an introduction to DVCS to demonstrate several different workflow scenarios that you can build with a DVCS. Which makes the important point that a DVCS can be adapted to your workflow in a way that is at best unwieldy with a CVCS. I intend, though, to break with tradition here.

By this stage, I hope you can see that distributing version control works by introducing branches where development takes place in parallel. Mercurial treats these branches as arising naturally from the commits made and transferred between repositories. Both Git and Bazaar take a slightly different viewpoint, and explicitly generate a fresh branch for work in a particular repository. But in both cases the underlying principle of identifying changes by a globally unique identifier and resolving parallel development by merges between overlapping changes is the same. And all three can be used in a truly distributed manner, with full history and the ability to commit being available locally.

So instead of chatter on about workflows, I want instead to reflect on the consequences all this has for that all-important question of whether a DVCS is a suitable vehicle for your data.

The first is a minor and rather obvious point. If you want to store files that are very large and which change often in your DVCS, then all the compression in the world is unlikely to stop the storage requirements for the full project history from becoming uncomfortably large, particularly if the files are not very compressible to start with.

The second, and main, point is that there is an important question you need to ask about your data. We've seen that a DVCS relies on branching and merging to weave its magic. So take a close look at your data, and ask:

Will It Merge?

The subset of plain old text which comprises program source code requires some human oversight, but will merge automatically well enough for the process to be well within the bounds of the possible.

Unfortunately when we move further afield mergeability becomes a rarer commodity. I nearly began the previous paragraph by stating that plain old text will merge well enough. Then Doubt set in — what about XML? Or BASE64 encoded content?

Of course, merge doesn't necessarily have to be textual merge. I am told that Word can be used to diff and merge two Word `.doc` files, a data format notorious for its binary impenetrability. As long as some suitable merge agent is available, and the DVCS can be configured to use it for data of a particular type[12], then there is no bar to successful

[12]Mercurial can have the merge and diff tools specified with reference to the file extension on which they

DVCS use.

Before this reliance on mergeability causes you to dismiss DVCS out of hand, reflect. A CVCS can only handle non-mergeable data by acting as a versioned file store; in other words, having as the only available merge option the use of one or other of the merge candidates in its entirety. Useful though a versioned file store can be, it cannot be considered a full-featured version control system. By treating the offending unmergeable files as external to the DVCS, or with careful workflow — disabling the distributed and mergeable potentials — a DVCS can deal with these files, but only at a cost of its distributedness or its version control system-ness. In this it differs little from a CVCS.

So, for all data you want to version control, let your battle cry be:

Will It Merge?

At this point, I have an urge to don lab coat and safety goggles and be videoed attempting to mechanically merge data in a variety of different formats. Frankly, this is unlikely to be as exciting at blending iPhones[13], but from a system development point of view it's rather more important. And, I think gives us a large clue as to one of the reasons for the continuing popularity of Plain Old Text as a source code representation mechanism.

operate — I assume Bazaar and Git are similar.

[13] http://www.willitblend.com