Living on the Edge – Automatic Merging

By Tom Tyler

The topic of fully automating merges comes up every now and again. By “fully automating”, I mean taking the human out of the resolve process entirely for changes that don’t require either an interactive resolve or advanced integration options. A changelist is submitted by a human on one branch, and that submit initiates one or more automated integrate/resolve/submit operations to one or more target branches along a predefined merge pathway that reflects your current branching strategy.

Eliminating humans from the process seems like a bold step. And indeed it is! Perforce’s merging algorithms are a “best guess”, based on what I call “text diff-chunk based logic.” It’s a tremendous value of Perforce that it does this so well, but there’s only just so far that it can take you. Perforce doesn’t try to understand the semantics of the C++ or Java or Perl code you’re writing, and it’s entirely possible that its best guess (the resolve result) won’t be what you want. Such merge errors might only be detected at compile time. The worst case scenario is that you introduce a “semantic merge” problem, where the result of an automated merge is incorrect, but compiles OK and sneaks past testing, possibly even surviving long enough to be a customer-visible bug. ***Eek!***

Because they can escape detection longer, semantic merge problems are harder to fix later, since the change is no longer fresh in the mind of the developer who originated the change. (It’s usually best to merge changes soon after they are made – ideally by the developer who originated the change).



Figure 1: The Risk of Automated Merging

Maybe you’re thinking, “This is too risky! Why would anyone ever do this?”  Visions of train wrecks pop in your head! But there are indeed reasons to do it! (To consider automated merging, not to wreck a train).

The risk of semantic merge problems is real, but there are considerable benefits to factor in when evaluating whether automated merging is right for your organization. The main benefit is ensuring fast propagation of changes. Automated merging helps ensure that once-fixed bugs never rear their ugly heads again. Besides, semantic merge problems occur even without automating merges if the human whose job it is to catch the merge problems misses them. So, while automated merging might increase the risk of having more semantic merge problems, it can also decrease other types merge problems, including the dreaded “repeat offender” bug.

Semantic merge problem risks can be reduced by automating merges only along pathways most likely to produce a good result. For example, you might automate merges from a mostly stable release branch used (almost) exclusively for bug fixes back to MAIN.

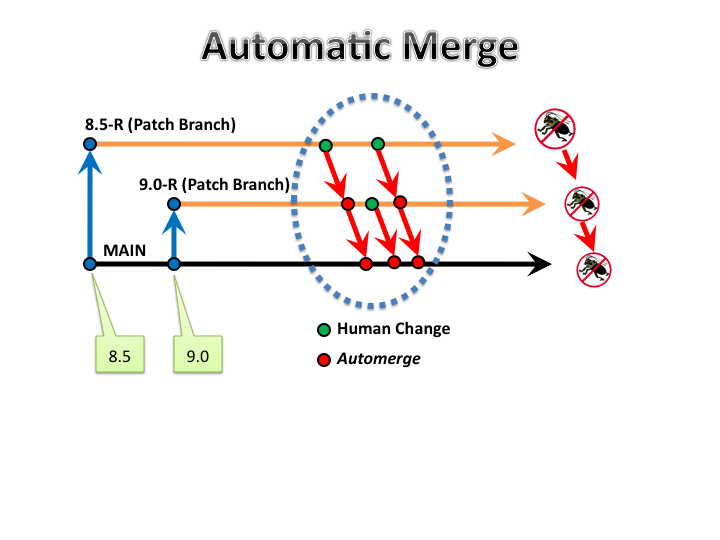


Figure 2: Automerge - Fast Bugfix Propagation

There are a lot of things to consider if you want to automated merges. Here are just a few:

* Define tag names for each branch, e.g. FGS-3.0-R = //Eng/FGS/rel/3.0-R/…
* Define merge paths between branches, e.g. FGS-3.0-R to MAIN.
* Consider which merge paths are most appropriate for automated merging.
* Provide a way for an admin to enable/disable each defined merge pathway.
* Provide developers a way to avoid automerging particular changelists (e.g. with special text in the changelist description).
* Sending email notifications when automatic merges occur, with a subject line indicating if it succeeded or if interactive (human) resolution is required, etc.

Automated merging works best if you automate merges from bugfix-only branches.  In more general terms, automated merging works best if the branch from which you’re automatically merging contains focused changes, tightly controlled changes.  Things like general code cleanup will make for a mess, so it’s best if that sort of activity occur in new development branches, not release maintenance branches. In branching strategies where products change extensively on long-lived release branches (as opposed to bugfix-only release branches), the success rate for automatic merges would likely be lower. Automated merging may not be appropriate in that case.

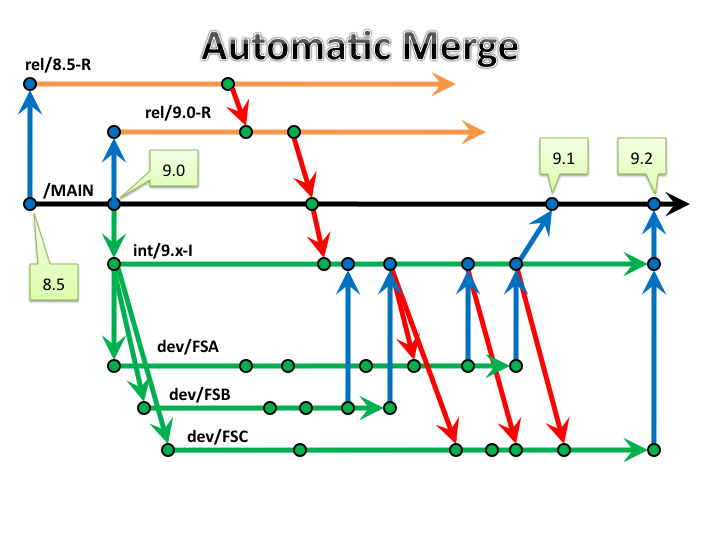


Figure 3: Automatic Merge Paths

**Which paths should be automated?**

Take a look at the diagram above. There’s a lot going on there! That depicts an advanced branching strategy involving a mainline, release branches (rel/\*-R), integration branches (int/\*-I), and development branches. Start with automating merges from the release branches to MAIN only. In the example above, the merge pathways from rel/8.5-R to rel/9.0-R and separately from 9.0-R to MAIN would be candidates for automated merging. The remaining merges would be left to humans.

Personal development sandbox branches aren’t shown to avoid clutter. If sandboxes are used, allow developers to automate merges from development branches to their personal sandboxes.

**Where does automation breakdown?**

Focusing automation only on paths likely to produce a good result is one way to help prevent merge automation from leading to chaos. But even when automated merging is helpful, there will still be some merges that just can’t be automated.

For example, an interactive resolve is necessary whenever there are conflicting chunks of text modified in the same file in both the source and target branches. Such merges require human interaction to resolve.

Advanced integration options are required scenarios where Perforce isn’t sure if a change should be propagated. For example, if a change is made to a file in the source branch, but the corresponding file in the target branch has already been deleted (or vice-versa), a human with an understanding of the history must determine the appropriate course of action. For these scenarios, a human should use the Revision Graph, Time Lapse view, and even the old fashioned telephone. Then resolve manually based on the results of analysis of the history, and possibly communication with others. The correct choice might be to supply the advanced integration option to Perforce, and proceed with the integration. Or it might be to modify the branch spec, adding an exclusion mapping telling Perforce to ignore further changes to certain files (such as those already deleted in the target branch but which need to exist in the source branch for some reason). [See the Perforce KB article, ‘[Preventing the Propagation of Deletes](http://kb.perforce.com/UserTasks/CodelinesAndBranching/PropagatingDeletes)’].