Branch Management and Atomic Merge in a Continuous Integration Environment

*Yi Zhang and Ray Chang
Information Intelligent Group, EMC Corporation
6801 Koll Center Parkway, Pleasanton, CA 94566

Abstract

An automatic merge tool was implemented as the centerpiece of branch management in a continuous integration environment. A naming convention and several processes are enforced with Perforce triggers to enable the connection between Perforce and the defect tracking system. The coupling of the "Tofu" branch model with the defect structure allows efficient management of merges and fixes across various branches. Update of defect IDs is automated as part of the atomic merge process and users can directly access source code from the defect tracking system. The combination of the wholesale daily atomic merge and an on-demand merge web interface has increased the merge rate, reduced branch divergence, and greatly enhanced productivity and quality. This paper also describes challenges encountered during the implementation and the remaining unresolved issues.

1. Introduction

To improve product quality and engineering productivity, EMC's Information Intelligence Group (IIG) has embraced lean Agile software development methodologies. All projects were migrated to iterative development three years ago. As a result, there were two immediate changes in engineers' day-to-day life: 1) Continuous integration replaced the regular daily build model. Engineers integrated and built more frequently. 2) Release cycles were shortened from months to weeks. Engineers were required to deliver GA-quality code within a short, four-week iteration.

Subsequent to methodology changes, we were confronted by several new challenges. First, we could not smoothly transition between two iterations without sacrificing team productivity. Second, multiple iterations exacerbated the issues of human-induced regression. It became increasingly challenging to determine which bug fixes were in a particular branch. This was true for branches within and across iterations, and especially between releases. Third, branches were becoming more divergent; it was almost impossible to perform meaningful merges. All these challenges stemmed from the fact that IIG, over the period of many years, had accrued a significant amount of debt (Sterling 2010) in the area of Software Configuration Management (SCM), and branch management in particular.

*Correspondence to: yi.zhang@emc.com

The organization had formed an institutional habit of anarchic branching, merge-phobia, and operating without a workable SCM process.

Migrating to iterative development made it painfully obvious that it was time to pay down the SCM debt. That is, to adopt an effective branch and merge practice. Otherwise, we would continue to incur significant loss in productivity and quality, making it impossible to succeed in a continuous integration environment. In section 2, we discuss the recommended branch model, and describe how to couple the branch model with the defect tracking system. In section 3, we discuss the P4PERL (Perforce Software 2010) atomic merge tool and the implementation details of linking the merge tool with the defect tracking system. Conclusions and discussion are in section 4.

2. Branch and version model

Fig 1 shows a typical version tree of a source code file from Perforce Depot. One common characteristic from most version trees is that all branches were created randomly and no merges were performed among branches. As a result, the source code branches diverge.

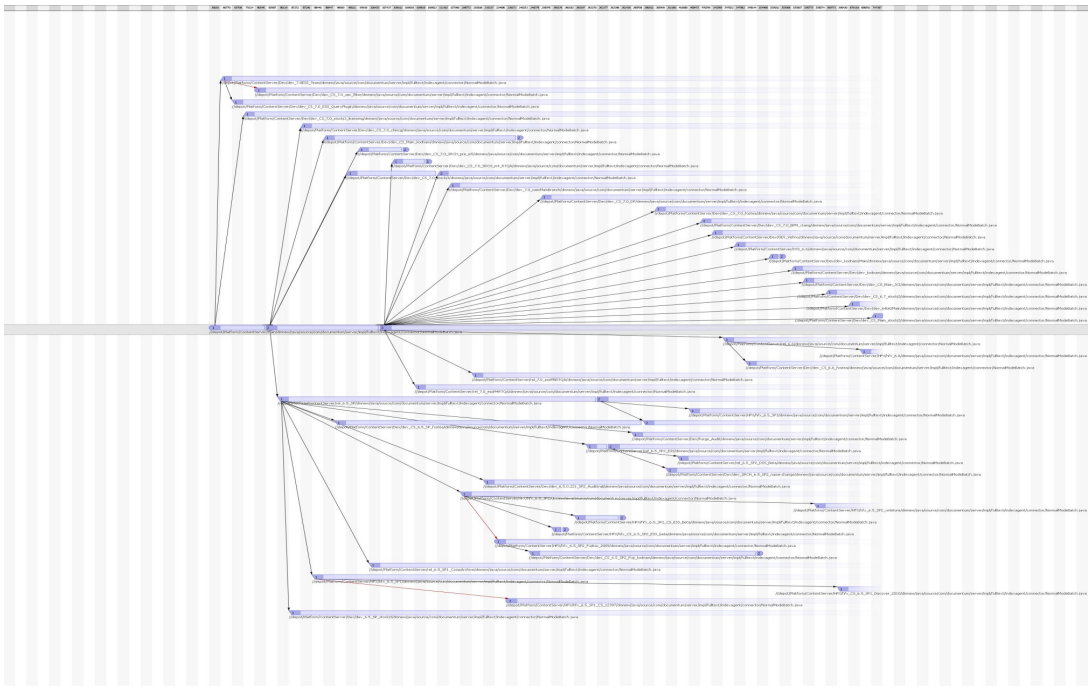

Fig. 1 A typical version tree of a diverged source file from Perforce.

Diverging branches have serious consequences in productivity and quality (Johnson 2009): Once the branch diverged over a period of time, merges became extremely difficult, and eventually impossible. As a result, engineers would have to check the same code into different branches, an error-prone manual process. Sometimes, engineers forgot to check files into all the

necessary branches, causing regressions. In the worst case, if the regression was not caught by QA, the released products were at risk of containing critical defects that would lead to customer escalations. Worst of all, when escalations occurred, there was no easy way to tell where the particular fix was checked in and where it was not. This was the most expensive type of regression we experienced.

As the first step of resolving this longstanding issue, we defined a set of consistent naming conventions across the source control and defect tracking tools. A 3-5 letter "Product Code" was used to uniquely identify a product in Perforce and the defect tracking system. This code was embedded within all

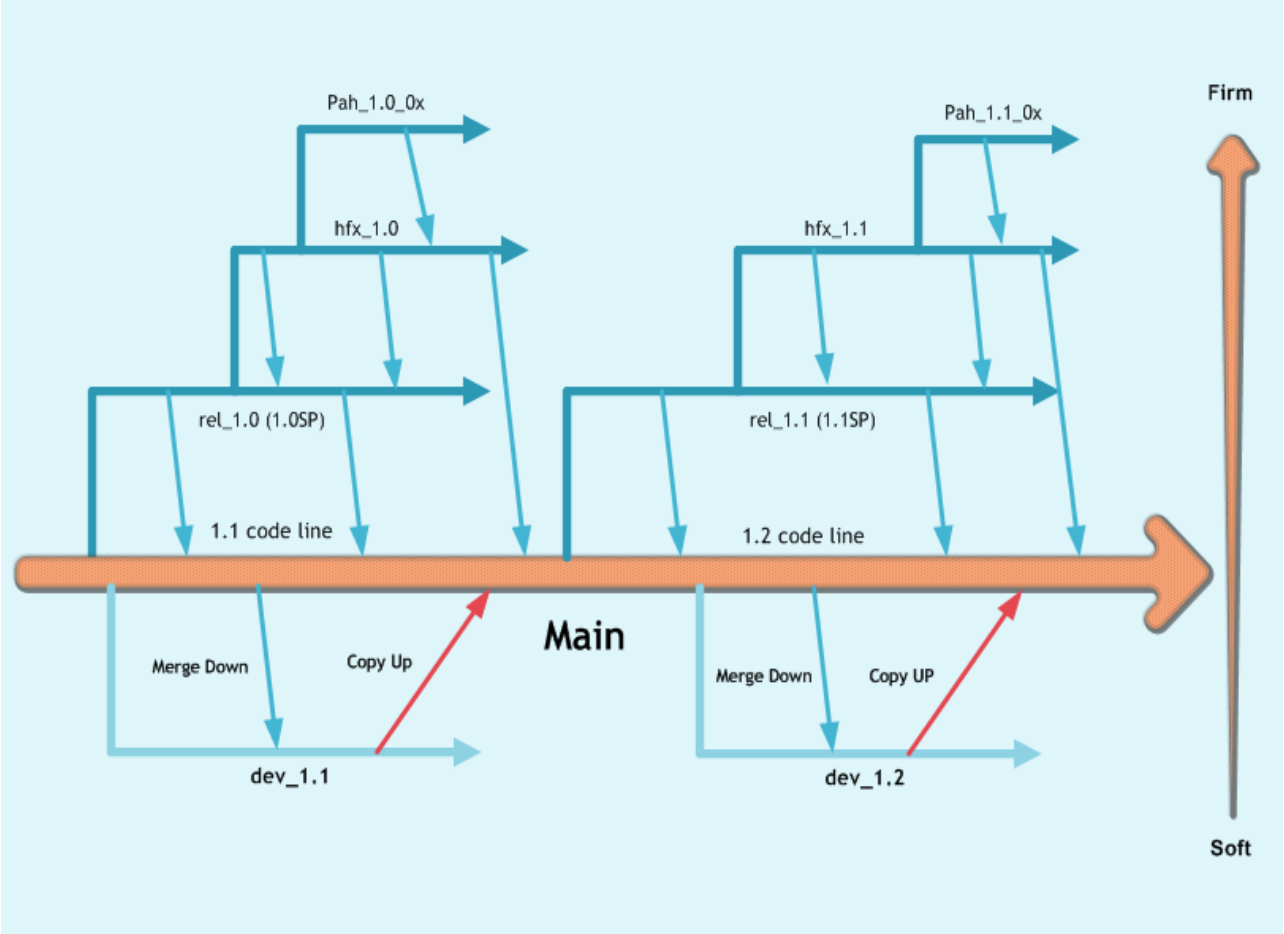

Fig. 2 Perforce branch model implemented in the context of "Tofu" following Wingerd (2005)

product-related artifacts such as defect ID, label, branches, etc. With this convention, the automation of processes and enforcement of policies became possible using scripts and Perforce triggers. We restricted branch creation to only five types and educated engineers to follow the "Tofu" scale, a branching best practice recommended by Wingerd (2005): "Firm on top, soft

on bottom; always merge down and copy up, never destabilize the code line by merging from a less stable code line into a more stable one."

Fig 2 shows the five types of branches in the life cycle of a product, portrayed in the context of the "Tofu" model. The types are: development, Main, release, hot fix, and patch, in the order of increasing firmness.

We encouraged engineers to merge down frequently, then presented branch diagrams to all engineers and explained how to merge in P4V. Take the example of the release 1.0 (rel_1.0) code branch; if there is a hot fix requested by a customer or QA, it is fixed in the "hfx_1.0" branch and merged down to Main. Patches are released monthly, and a patch branch "pah_1.0_0x" contains the accumulation of hot fixes within the month. Any changes requested on top of a patch are fixed in the patch branch and merged down to the hot fix branch. All the changes in the hot fix branch "hfx_1.0" are merged into the release branch "rel_1.0," which is also where service packs are released. A service pack contains all the patches and hot fixes accumulated up until release time. The patches and hot fixes should also be merged down to the Main branch where the current development is happening. In a perfect world, every fix is propagated down by either auto or manual merge, and the next release branch always contains everything in the previous releases.

To facilitate this branch model and help engineers to form the correct mental picture of the relative firmness of all branches within a project, we strongly recommended that all branches should be put at the same directory level, and organized similar types within one folder. As indicated in Fig. 3, "HFX" contains all hotfix and patch branches, "Dev" contains all the development branches, release branches are put in parallel with the other branch type folders. This way, by opening P4V, engineers know exactly which branch type belongs to which folder. This branch structure organization made it much easier for engineers to follow the "Tofu" branch model.

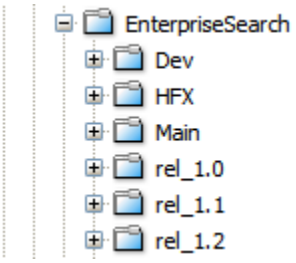

Fig. 3 Perforce branch structure within project "Enterprise Search".

Setting up a proper branching and merging model was not enough: we needed to record the details of a code fix, its location, status, and direction

of propagation. For this purpose, we introduced a new defect model. This model is tightly coupled with the "Tofu" branch model. A new type, "child fix request" was introduced to record a fix in a particular branch. Fig. 4 shows the structure of a defect. There are four children under this defect, each one is called a "child fix request", and each one corresponds to a specific branch in Fig 2. For example: Fix Request 3 is associated with hot fix version 1.0SP2Hotfix. This means the fix will be checked in into the hfx_1.0 branch in Fig. 2. The Development branch does not contain any bug fixes; therefore, it does not have a corresponding child in the defect family. The parent defect is a placeholder, which has a description field and two states: "Open" or "Closed". A defect can only be marked "closed" when all the children are closed. An important constraint is enforced, which requires a child fix request to be created in the current main code line with the creation of a defect. This ensures that any upstream fixes are included in the current code line. In theory, any human-induced regression is recorded and should be completely avoided. Even if it occurs, we have a mechanism to know the exact branches a particular fix was missing.

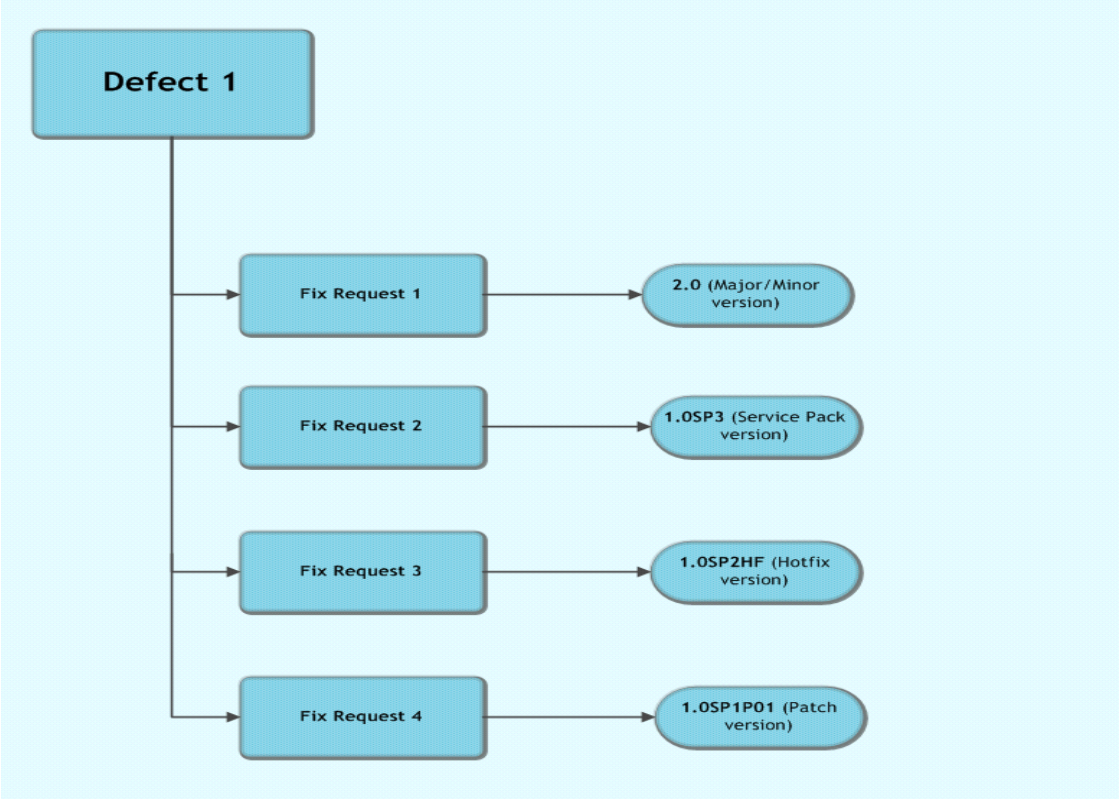

Fig. 4 Schematic of a defect and its children, each child corresponding to a branch in Perforce.

The introduction of the new branch and defect model was a major cultural change within our organization. Reactions from Engineering varied from acceptance, to indifference, to opposition. For some teams, the migration to Perforce saturated their tolerance for change: they were simply not ready to embrace the additional process changes associated with the tool migration. For most engineers, the new process and model "made good sense" at the training session but was quickly forgotten afterwards. The SCM team put a lot of effort into explaining the new process and model. We faced numerous issues and obstacles during and after migration to this new model. It was soon apparent to us that setting up the branch model and new defect type was only the first step, and the easiest step of all. The most difficult task was to change the institutional habits that our organization formed over many years. Instead of fixating on how "wrong" these habits were, the SCM team chose to automate the merge process and its integration with the defect tracking system. Our motivation was simple: in the long run, if we could make engineers' code-writing experience easier and more efficient, they would naturally follow the recommended best practices.

3. P4PERL auto merge tool

We started by developing a simple automated merge tool to perform recursive merging from the hot fix and patch branches to the release and main branch. This tool was developed based on P4Perl, inspired by the "findmerge" utility of IBM Rational's ClearTool (Rational Software 2000). Implementation of this tool was intended as an important starting point for changing engineers' habits. We wanted to encourage engineers to start merging and have it become a habit. This way, we are able to reduce human-induced regressions and branch divergence. This auto merge tool is scheduled to run daily on a designated machine. For non-conflict merges, a one-time email notification is sent out to all stakeholders regarding the merge result. For merges that require manual intervention, a daily email is sent to the source and destination code owners until the merge conflicts are completely resolved.

We first ran this tool between a Main and a development branch. It was quite successful. Nearly all the merges were automatic, because merges happened every night. However, when applying merges from the patch and hotfix branches to the Main branch, engineers raised many concerns and issues. Among them was the issue that merges broke the atomic set. For example, a previously fixed bug contained five files, but only three files were merged due to conflicts in the remaining two. This broke the compilation. It also caused confusion on the defect tracking side, because engineers didn't

know how to mark the status when a fix was half-way done. Furthermore, it broke the link between the original defect and the Perforce change set.

This caused quite a setback in the promotion of this new merge tool and branch model. We quickly improved the tool by integrating atomically. We still use "p4 integrate" to compile the list of files that need to be merged. Once the list was available, it was sorted against all change list numbers. We would then merge the entire set only if all files within the set could be merged automatically. After one week of production testing, this new modification was well received in the user community, and the atomic approach completely replaced the previous file-level merges.

We also added two more options for "p4 integrate." We provide users the option of excluding folders and files under a directory tree in the source branch. This is quite useful for relatively static projects such as localization, build, and Tech Pubs. We also added the option of excluding all files that submitted by certain users such as build accounts.

The logical flow chart of the atomic merge tool is shown in Fig. 5. The input of the merge tool includes the source and destination branch depot paths, the file and folder exclusion list, and user exclusion list. Within P4Perl, we call the "p4 integrate" command with the "-d" and "-i" option (Perforce Software 2007), and generate a complete list of files (thereafter file-set) to be integrated between the source and the destination. To prepare for the exclusion of the file-set, we first mark the files that are last modified by the users on the user exclusion list, and then mark files that are on the folder and file exclusion list by recursively comparing all the files with the exclusion files. Then, we invoke "p4 resolve" on the subset of unmarked files, and mark all files that have merging conflicts. At the same time, we compile a complete set of change list numbers, based on all the files in the file-set. Next, we sort all the files by the set of change list numbers. We loop through all the change list numbers and divide the change list into three categories: First, the change list contains at least one conflict file – a file that requires manual intervention. We run "p4 revert" to revert all the changes made on all the files on the change list, identify stakeholders of all the conflict files, and send out daily email to the stakeholders until the conflicts are resolved and the merge is performed. Second, the change list contains at least one file that must be excluded. In this situation, we also revert all the files on the change list.

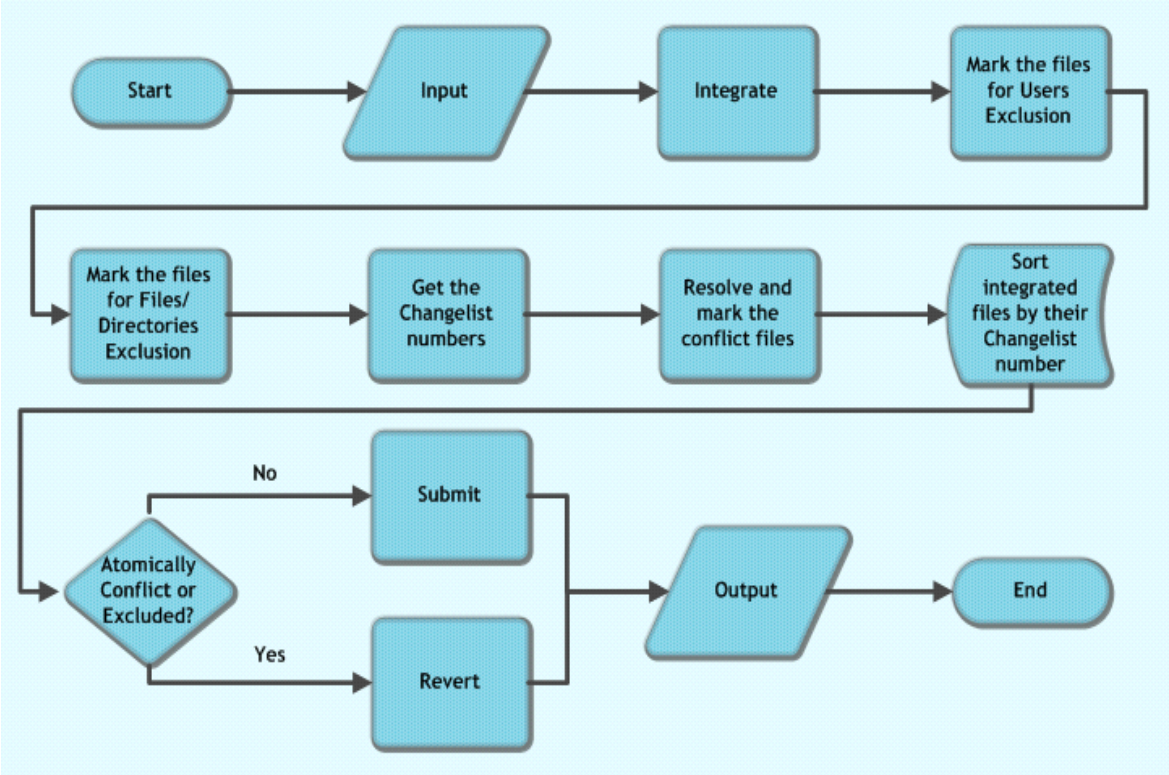

Fig. 5 Logical flow chart of the atomic merge tool based on P4PERL

Third, if all the files within a change list can be resolved automatically, we run "p4 submit" and commit the merge. Once it is done, we send an email to the stakeholders announcing the status of the merge. The situations that a change list contains both conflict and exclusive files belong to the first scenario.

The atomic merge solves a very important practical issue: to efficiently propagate code fixes among different code lines. The atomic merge is based on "p4 integrate," integrating at the level of a single file. For this reason, the atomic merge is not strictly at the change list level. Instead, the tool is based on the following assumptions. First, change lists are restricted within one branch tree. Any change list that spans multiple code lines is considered invalid by the tool. Second, merges only occur among the tip even if a file is modified multiple times between merges. For this reason, if two change lists share a number of files, the first one will always contain fewer to-be-merged files because the shared files are considered in the latter. Third, breaking of the atomic rules is allowed if the change list contains exclusion files. All the assumptions proved to be necessary to satisfy the practical use cases, at least none presented a limitation. Because we only propagated fixes among branches, we have not seen any change list across a code line. The first assumption has always been a valid one. For the second assumption, even

though the merged change list might contain fewer files, it never caused compilation issues because the second list always takes care of all the changes in the shared files with the first list. The third assumption proves to be necessary. This is because broken change lists that contain exclusions at both the user and file level are helpful for some projects. In the end, even the merge tool is not strictly atomic; it serves well to solve practical issues.

An important improvement and logical step that is not described in the flow chart is the automation and integration with the defect tracking system. To be precise, the linkage between an atomic merge set and its corresponding fix request in the defect tracking system. Because we enforced the rule that every check-in must be associated with a fix request ID in the defect tracking system, we were able to automate the entering of the correct fix request ID associated with each successful automatic merge.

This process is illustrated in Fig. 6. The dotted line with the two-way arrows indicates the two-way linkage between Perforce and the defect tracking system. Prior to check-in of the merged change list, we examine the check-in comment of the source branch and identify the fix request ID(s) in the comment. Once the fix request ID is identified, we search the IDs of the sibling fix requests from a set of data retrieved from the defect tracking system. Based on the destination branches in the merge request, the correct ID of the fix request is identified and entered in the comment of the merged change list.

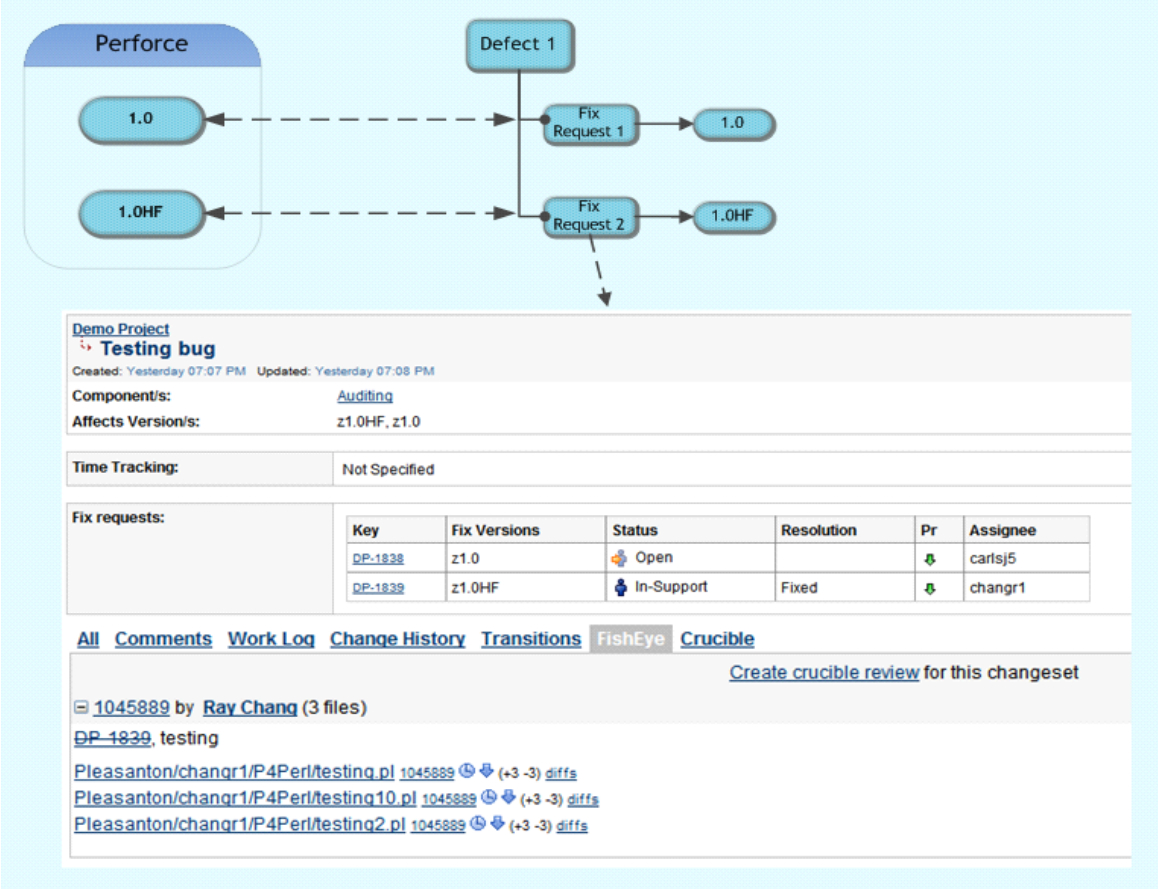

Fig. 6 Schematic identifying the correct fix request ID, and entering the ID in the check-in comment of a merged change list. This way, the merge tool automatically enables the linkage between Perforce and the Defect-tracking system. Users are able to access the source code change list associated with a fix request directly from the defect tracking system.

Once the merged change list is committed and the fix request ID included in the check-in comment, the source code, the check-in comment, and the code review comments become automatically accessible from the corresponding fix request in the defect tracking system, as indicated in the lower panel of Fig. 6.

Another feature added to the original merge tool is a web interface. The wholesale automatic merge does not apply to all the projects and branches at all times. Engineers often need to request ad hoc merges. With this web interface, engineers must enter Perforce user ID, source and destination branch depot paths, and optionally a change list number. The tool automatically merges the changes into the multiple branches specified, and marks the correct sibling fix request IDs in the merged change list. Engineers are able to access the source code of a fix request from the defect

tracking system. Similar to the wholesale approach, requesters receive emails and might need to resolve conflicts manually.

We integrated our merge tool with TeamCity, a continuous integration build automation tool, to leverage its scheduling and queuing capabilities. We set up triggering of builds based on a successfully completed merge. Within TeamCity, an automated sanity test was scheduled after the merged build. Status of the build and test was also sent to the corresponding stakeholders. We intend to augment the post-merging process by running quality tools such as "findbug".

4. Conclusions and discussion

Before we started this project, we had several end goals in mind: to completely eliminate unnecessary manual work, to eliminate human-induced regressions so that we can be unequivocally confident that all downstream code lines contain every necessary change of previous releases, and to minimize branch divergence.

Our strategy was to use the auto merge tool as a starting point to alter engineers' old habits. This merge tool was in production for three projects: one project was in its initial development phase, another project was in a final release, and the third one was in the middle of development. During a three-month period, the merge tool performed over 1,500 change lists automatically with success for the first project. The tool also performed all bug fixes (close to 100) from release to Main for the second project. In the third project, more than 1,400 change lists were automatically merged. Overall, the automatic merge comprised over 95% of all merges. We felt that we made an important and very successful first step towards our goal.

We are still facing many issues. For example: we cannot find an effective way to prevent engineers from merging up. Some engineers are used to working in the Main branch, and switching out of main requires additional work; they would rather check their code into Main and later merge into a release branch. We have to monitor the branch and can only find the issues once they have occurred.

Sometimes, engineers check the same code into several branches instead of merging. When this happens, Perforce, in many cases, cannot detect that the same change went into different branches. We had to ask engineers to merge using "p4 integrate" so that Perforce would recognize that the same code was contained in all branches. This became an issue for verification, because we were not able to depend on the output of "p4 integrate" to verify

that check-ins in a particular branch made it into the other branches. One of the items on our wish list is that "p4 integrate" could more intelligently detect the same piece of code in multiple branches.

Even with the exclusion of files, folders, and users, we find that there are exceptions we must include, such as a subset in an exclusion list or exclude files for a one-time merge. For example, a build account must merge a set of dependent binary files from one branch to another; engineers must make a one-time fix of platform dependent issues that cannot be merged from release to Main. In rare cases, we have to revert some of the merges performed by the auto-merge tool due to these exceptions.

The toughest issue we have faced is that certain teams and engineers refuse to follow the recommended process in branch management and merging. The reasons are too complex to analyze: some engineers might be afraid of merges, others simply don't want to change their old habits. We do not have a good answer for what to do with these situations. So far, what we have tried is to slowly, project-by-project, take control of release branches, monitor check-ins, set up the automatic merges, and put effort into explaining the best practice and helping engineers to follow the process.

We have significantly improved our branch management following the implementation of the auto-merge tool. We still have much work to do in order to reach our end goal. Under ideal situations, whenever we run a "p4 integrate" command between hfx_1.0 and Main and between rel_1.0 and Main (See Fig. 2), with the exclusion files and users, the returning result should be: "all revision(s) already integrated." If this message does not appear, the question should be: "what is missing?" When we first branched off the next release code line from Main, the "p4 integrate" command, with the exclusion files and users, showed anywhere between a few hundred to over one thousand files that needed to be integrated between the branches. Unfortunately, it was extremely difficult, if possible at all, to verify that all the changes were included in the release code line, because the branches were so divergent. In this case, we really had no choice but to rely on QA to test every single feature and bug that had been tested in the previous releases, and to pray that the difference at the code level did not introduce any significant issues on the released products.

We initiated this work because we saw enormous potential to boost productivity and quality. The old habit of anarchic branching had always resulted in a great deal of confusion, frustration, and unnecessary work for development, QA, and others in the product team. The introduction of the auto-merge tool, and the processes and automation around it, was a very important first step towards the improvement of productivity and quality.

Based on what we have observed since the implementation of the auto-merge tool, the entire development community is moving in the right direction. We are cautiously optimistic that we will be implementing our merge tool to more projects within the company. We will eventually reach a tipping point when the institutional habit will be completely transformed into "Following the Tofu model, prudently using the auto-merge tool" instead of "anarchic branching and merge-phobia."

Acknowledgement: We are indebted to Dennis Dawson and Michael Ottati for a thorough review of an earlier draft. Comments, discussions, and support from Randall Fong, Sameer Kachare, Henry Liu, and Jonathan Stockley are greatly appreciated.

References

Johnson, A. 2009: Visualizing the cost of divergence. Perforce User Conference.
Perforce Software 2007: Perforce 2007.2 Command Reference, 276pp.
Perforce Software 2010: Perforce 2010.2: APIs for Scripting (http://www.perforce.com/perforce/r10.2/manuals/p4script/index.html).
Rational Software 2000: *ClearCase Reference Manual* (A-L), 538pp.
Sterling, C. 2010: *Managing Software Debt*. Addison-Wesley, 244pp.
Wingerd, L. 2005: *Practical Perforce*. O'Reilly's & Associates Inc, 336pp.