FLOSS for safety: Mastering mission critical development with GIT

Andreas Platschek
Opentech EDV Research GmbH
Augasse 21, 2193 Bullendorf - AUSTRIA
andreas.platschek@opentech.at

Nicolas McGuire
Opentech EDV Research GmbH
Augasse 21, 2193 Bullendorf - AUSTRIA
der.herr@hofr.at

Abstract

Writing code for safety critical systems implies a lot of constraints and requirements in the software development process. Depending on the field a specific development process has to be followed and fully documented. In order to be compliant with a standard ,e.g. DO-178B, ISO61508, ISO26262, MISRA-C, stringent traceability requirements must be met, allowing to justify the development process it self.

In this paper, we try to give some examples of features of git that are - in our eyes - useful in helping the developers to fulfill the traceability and documentation requirements of safety related development life-cycles for bespoke components. Examples for such techniques are a history for each and every line of code, or the sign-off and ack mechanism, to find out who is responsible for any part of the code base - providing a method to ensure you know who to blame.

These git specific features can even be improved by using git hooks in combination with different tools, including formal methods, and backend scripting, allowing to fully automate these QA related extensions.

The intention behind all of these techniques is to build a toolchain for extended traceability [version control] around git, simplifying the verification process.

Although this paper targets safety critical systems in particular, some of the presented techniques may well be suited for enhancing the Linux kernel development as well as for standard user space programing tasks.

[DISCLAIMER:] None of the proposed techniques replaces common sense, they are just a way of improving your code and development process.

1 Introduction

GIT [1], the "information manager from hell" is a version control system. The project was started by Linux Torvalds out of the need for an FLOSS version control system for the Linux Kernel, which was capable of distributed development. Before GIT, the Linux Kernel development was based on BitKeeper a proprietary version control system, which was available for free use (here: "free as in free beer"), but got restricted in 2005 and therefore was no longer available for the development of the Linux Kernel. Since the FLOSS version control systems, such as subversion or CVS were no option either (mainly due to their centralized approach), Linus designed GIT which became self hosted in April 2005. Shortly after this, GIT was already used to host the linux kernel (starting from linux-2.6.12-rc2).

The main properties that GIT provides, are - as already mentioned - distributed development (details follow below), complete history, revision tracking and branching and merging are fast and easy. Furthermore no network access or central server are mandatory during development.

The remainder of this section is going to give you a short comparsion between centralized and distributed development, and discuss some requirements on the software development life cycle, introduced by popular standards such as MISRA-C, DO-178B, ISO-61508, ISO-26262 and others, that we think could be enforced, or at least be supported by git features and/or git hooks.

After this git hooks are introduced, and their capabilities as well as their limitations are discussed. The last section gives some simple examples which could have a big impact on the development of your mission critical software.

The more practical part of this paper requires a basic knowledge of GIT. If you are unfamiliar with GIT, you can find many tutorials and books (e.g. [2]) on the internet.

2 Centralized vs. Distributed Development

In Centralized RCS (revision control system) systems (i.e. subversion, CVS, ...), there is always one master repository (server). When a user checks the repository out, the local copy only contains the tip of the repository. Everytime the user wants to checkout a different branch, or an older version of the repository, he has to connect to the master repository and fetch the desired version.

In contrast to centralized revision control systems in GIT every copy of the repository is the same, this means, that in contrast to centralized RCS , you do not just checkout the tip of the repository, but you clone the whole repository. Therefore, after a clone, the new tree is exactly the same as the original (hence: clone).

This sounds like, a clone would take a very long time, but the good news is, that a clone is just marginally slower than a svn checkout. If you are interessted in a comparison between RCS systems in use, you might want to look at [8].

So the big question now is, what is the big advantage of having an exact clone of the repository? First of all obviously if you decide to have a dedicated server (often called *depot* in git), and for some reason the repository on this server gets corrupted, all you have to do, is to clone the local repository from a developer, and you are back in buisness.

Another advantage is, that you do not need a network connection to the server, when you are working, since you can commit into your own, local repository while beeing offline, and just push all those commits to the depot the next time you have a network connection.

Furthermore, this distributed approach, where every repository has the same priority allows you to implement workflows other than a centralized server where everybody commits to. In example, you could want a hierachical workflow (Dictator and Lieutenants Workflow), where the maintainer of the main repository (the dictator) pulls from a limited group of people's repositories (lieutenants), who pull from the repositories of all the "normal" developers.

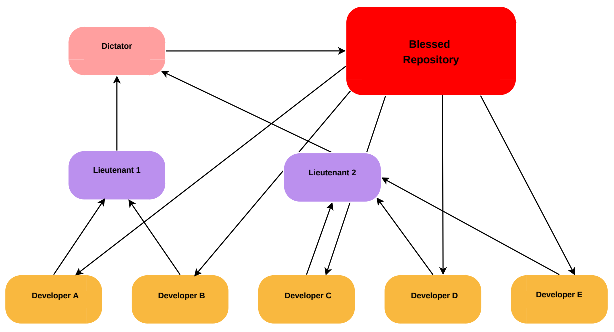

FIGURE 1: *Dictator and Lieutenants Workflow*

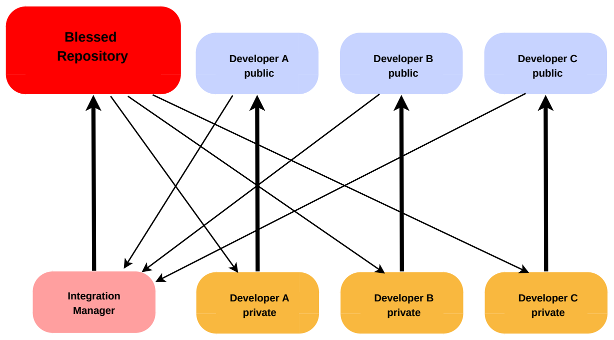

FIGURE 2: *Integration Manager Workflow*

In git there really are no real limits to the workflow, it really is just a matter of how you want to do it, and which approach fits you project the best. After all a "main" repository (or depot) is just a convention in git and technically does not differ from any other clone of the repository.

3 Requirements for Mission Critical Software

3.1 What Standards say about that matter

Before we look at requirements that can be assured with the proposed techniques, we will have a short look at what widely used safety standards say about the development of mission critical software.

IEC 61508 *a) apply administrative and technical controls throughout the software safety lifecycle, in order to manage software changes and thus ensure that the specified requirements for software safety continue to be satisfied;*

b) guarantee that all necessary operations have been carried out to demonstrate that the required software safety integrity has been achieved;

c) maintain accurately and with unique identification all configuration items which are necessary to maintain the integrity of the E/E/PE safety-related system. Configuration items include at least the following: safety analysis and requirements; software specification and design documents; software source code modules; test plans and results; pre-existing software components and packages which are to be incorporated into the E/E/PE safety- related system; all tools and development environments which are used to create or test, or carry out any action on, the software of the E/E/PE safety-related system;

d) apply change-control procedures to prevent unauthorized modifications; to document modification requests; to analyse the impact of a proposed modification, and to approve or reject the request; to document the details of, and the authorisation for, all approved modifications; to establish configuration baseline at appropriate points in the software development, and to document the (partial) integration testing which justifies the baseline (see 7.8); to guarantee the composition of, and the building of, all software baselines (including the rebuilding of earlier baselines);

[IEC 61508-3 Section 6.2.3.]

DO-178B Section 8.2 Software Quality Assurance Process Activities: *"The SQA process should take and active role in the activities of the software life cycle processes, and have those performing the SQA process enabled with the authority, responsibility and independence to ensure that the SQA process objectives are satisfied."*[DO-178B, Section 8.2]

3.2 Generic Description

Coding Style - There are various reasons why defining a coding style that is followed throughout a project is a good idea. In example, reading code of co-developers becomes much easier, if everyone uses the same indentation. Below in the examples section, we will use the checkpatch.pl script, used to enforce the linux kernel coding style [3].

Coding Guidelines typically include:

- line length restrictions
- naming conventions
- identation
- source code documentation standards
- banishing of ambiguous expressions
- ...

One very famous coding guideline that is very widely used in the automotive industry is MISRA-C [4], and checking against a MISRA-C checker would be a perfect use-case of the techniques proposed in this paper (see section 6.1).

Quality Assurance - Since quality is a rather subjective construct, defining an appropriate assurance of the same is not a simple task to do. In most cases quality is assured by periodic reviews, using various tags, GIT can be used to keep track of which revisions have been reviewed and by whom. g

Test Applications - RCS systems are often used to do a nightly build. In section 7, we are going to present a simple way to extract more information from such a nightly build than just "build failed" or "build succeeded".

Formal Checks Recently, various formal tools are used to find bugs in the Linux kernel. Those tools, like e.g. static code checkers can be used, to find potential threats periodically, and report their appearance and disappearance in a database.

Force Common Toolchain - Forcing all developers to agree and to use a common toolchain is - especially for FLOSS projects - impossible to achieve. To make sure that a commited version is compilable with the common toolchain, a server side git hook (for details look at section 5) that compiles the commited version with the official toolchain and rejects the commit into the common repository, if the compile fails.

4 GIT features

This section discusses some features provided by GIT that support you to ensure the requirements demanded by the standards you follow. Most of these features are going to be used in some of the practical examples, we are going to present below.

git log of course git offers the possiblity to browse through the history of development. The basic capabilities provided by git log (there is also a graphic tool called gitk) are extended by git blame, git diff, ...

git blame git blame shows you on a line basis who edited this line last. So you can reliably find out who to blame if you found the responsible person (I guess the times where one person is elected to be the scapegoat are now offically over...).

git bisect is a very powerful tool to find issues. Let's assume your software contains a feature that you know as already worked in some commit ABC, and you are at commit XYZ and for some reason, this feature does not work. Instead of testing every commit between those two, you can now just use git bisect to find the issue. What git bisect does is to checkout the commit exactly in the middle between ABC and XYZ, say MNO then you can check if your feature works in this commit. If it works, git bisect next checks out the commit exactly in between MNO and XYZ, if it does not work it checks out the commit between ABC and MNO. Using this technique you will find the last version where your feature worked very fast. Especially if it is an error that can be checked automatically.

git tag Tags are a very useful took, to mark commits, in example a commit could be tagged with "review_april_1_2010". At the next periodic review, there won't be any doubt which version already has been reviewed.

Signed-off-by Git allows to tag patches with tags, describing who has been involved with the patch. These tags include Signed-off-by, Acked-by, Tested-by, Reviewed-by, Reported-by and Cc. Although these tags have very specific meaning in the linux kernel tree[5], this meaning is only a convention, and could be different within your software project. Nevertheless, these tags are a great tool to describe a chain of people associated with a patch.

git diff can be used to easily show the difference between an already reviewed version of the development tree and the current version.

5 GIT Hooks

GIT Hooks provide an easy way for everyone to extend git to their needs. There is a number of different hooks available, (just `cd` into your repository and do a `ls .git/hooks` to list them all) the difference between all those hooks is when they are called - or in other words, the action that triggers them. For example the pre-commit hook (which we will be using later in the examples) is called every time before you perform a commit. A pre-commit hook offers the possiblity to run tests on the patch/the new version before you actually add it to the repository. Other hooks, like the post-receive hook are more suitable for depots, where they could e.g. be used to send an e-mail to a group of people everytime some pushes to the repository (a great way to produce spam...).

To implement a hook, all you have to do is to implement the hook (obviously), put it into the `your-git-repo/.git/hooks`, name it properly (i.e. according to the event that should trigger it, e.g. pre-commit. Usually you already have a sample for every possible trigger in your `.git/hooks` directory after initialization.) and make it executable (e.g. `chmod +x pre-commit`).

6 Simple Examples

This section contains some very simple examples. They are simplified and do not cover all possible commit scenarios. Most of them are restricted to C-Code only commits, so if you want to use them, you will have to extend them to your needs.

6.1 Coding Style

The very first git hook that we experimented with is a very simple pre-commit hook, that runs `checkpatch.pl` against the patch that is commited (the diff between the last commit and this commit), if checkpatch returns an error, the patch is rejected, warnings are displayed, but do not lead to a rejection of the patch.

As mentioned above, `checkpatch.pl` is a script that checks the linux kernel coding style. You can find the script in the source of the linux kernel in the directory `scripts`.

```
#!/bin/sh
git diff --cached > .git/hooks/tmp/diff.txt
.git/hooks/./checkpatch.pl --no-tree \
    --no-signoff .git/hooks/tmp/diff.txt \
    > .git/hooks/tmp/out.txt

cat .git/hooks/tmp/out.txt

if grep "ERROR" .git/hooks/tmp/out.txt; then
    echo "Coding Style Errors! COMMIT \
                     REJECTED!" >&2
    exit 1
fi

exit 0
```

The advantage of running the script against the diff, is obviously the speed, since you do not have to check the whole code base, but only the current commit.

Of course, there could be an exception, where breaking the coding style is inevitable, in such a case it is possible to add a `--no-verify` to the `git commit`. Of course this simple example could be rewritten to use a MISRA-C checker very easily, and could therefore be used to enforce the MISRA-C rules from the very beginning of the project.

6.2 Avoiding Complex Patches

The next example will deal with the avoidance of complex commits. This is done using cscope to build a (partial) callgraph. The Hook checks if the functions patched in this example call each other (this means, if you want to patch f1 and f2 and f1 calls f2 or f2 calls f1), and are therefore directly dependent. If they are directly dependent, the commit is rejected.

```
#!/bin/sh
funcs=$(git diff -p -U0 --cached | egrep ^@@ | \
     sed 's/(/ /g' | awk '{print $6}' | uniq);
echo $funcs

for i in $funcs; do
   tmp=$(cscope -R -L3 $i | awk '{print $2}')
   for j in $funcs; do
       if [ "$i" != "$j" ]; then
          out=$(echo $tmp | egrep "$j")
          if [ "$out" = "" ]; then
            echo "checked function"
            echo $i
          else
            echo "dependent! COMMIT REJECTED"
            exit 1
          fi;
       fi;
   done;
done;
echo "independent!"
exit 0
```

A completely different way to avoid complex patches - which we won't show here - , could be by using software metrices.

7 Automated Nightly Tests

Many FLOSS projects use nightly builds to provide binaries with the latest features for testing to the users. The example given in this chapter has a different aim, namely to check whether the current version is even compilable, and in case is not, to find out which commit broke the build process, and who is the person responsible for that patch.

Furthermore, these automated nightly builds are server side scripts, and can therefore also be used to assure, that the tree builds with a certain toolchain.

7.1 Finding bad builds

Basically this means, that a Testmachine (Testcluster) pulls the repository during the night and tries to build the last commit. If the build succeeds, the testmachine is done for that night, else it performs a `git bisect` on the last commit and the commit where the testsuite succeeded last (called `last_good` in the code). This allows the testmachine to throw back more than just "BUILD SUCCEEDED" or "BUILD FAILED", but also the exact commit where it stopped to succeed and the developer responsible for this commit. The use of `git bisect` allows the script to find the point where the build process fails very fast, even if many commits have been made during the day.

```
#!/bin/bash

# BEFORE FIRST USAGE:
#
# create a file last_good.one and copy the
# hash of the newest commit considered as
# good into this file. The hash will be
# updated every time this script is run.

# kill possible aborted bisect!
```

```
git bisect reset master
rm run_check.log

finish_flag=0
last_good=$(cat last_good.one)
echo $last_good

cp config_guest .config
make -j4 bzImage #2&> /dev/null
ret=$?

if [ $ret -eq 0 ]
then
   make distclean
   echo DONE.
else
   make distclean
   git bisect start HEAD $last_good \
         > run_check.log
   cat run_check.log

   while [ $finish_flag -eq 0 ]
   do
      cp config_guest .config
      make -j4 bzImage #2&> /dev/null
      ret=$?
      if [ $ret -eq 0 ]
      then
         git bisect good >> run_check.log
         cat run_check.log
         if (cat run_check.log | \
               grep "is first bad commit")
         then
            finish_flag=1
         fi
      else
         git bisect bad >> run_check.log
         cat run_check.log
         if (cat run_check.log | \
               grep "is first bad commit")
         then
            finish_flag=1
         fi
      fi
      make distclean
   done
fi

cat run_check.log
first_bad=$(cat run_check.log | \
   grep "is first bad commit" | \
   awk '{print $1}')
git checkout $first_bad

# print information about last good/first bad
git log -n1 HEAD \
   --pretty=tformat:"First Bad Commit: %H"
git log -n1 HEAD --pretty=fuller
git log -n1 HEAD \
   --pretty=tformat:"Last Good Commit: %P"

# also dump it into the .log file
git log -n1 HEAD \
   --pretty=tformat:"Last Bad Commit: %H" \
   >> run_check.log #last good one
git log -n1 HEAD --pretty=fuller \
   >> run_check.log #last good one
git log -n1 HEAD \
   --pretty=tformat:"Last Good Commit: %P" \
   >> run_check.log #last good one

git log -n1 HEAD --pretty=tformat:"%P" \
   > last_good.one #last good one

git bisect reset master
git checkout master # go back to tip
```

7.2 Finding Potential Threats with Stanse

One example we implemented, uses the static code checking tool stanse [9], to find potential threats like memory allocation errors or bad locking disciplines.

Stanse has already proven it's usefulness by finding dozens of bugs in the linux kernel (for more information, have a look at the project's homepage [9]).

In this example, we tried to automate the process of finding new (potential) bugs on the example of the linux kernel. The workflow of this example is as follows: The repository is pulled periodically (e.g. every night). After that, stanse runs a variety of different checkers over the whole tree, generating XML files with all the potential threats found. This XML file is then read by a python program, and all threats that have appeared for the first time, are added into an SQL database, and a list of new found threats is sent to the maintainer per e-mail.

In addition to the information provided by stanse, the database will include the hash of the commit where the threat was found first, a status flag, indicating the state of the threat (NEW, BEEING PROCESSED, RESOLVED), a hash of the commit where the threat is resolved as well as the possibility to write a note about the threat (e.g. "why this a false positive", or "what I tried so far to resolve it, but did not work").

8 Conclusions

In the words of DO-178B: the quality assurance process should play an _ACTIVE_ role in the development of safety/mission critical software. This paper showed some basic techniques how git can be used to easily implement mechanisms which not only support the quality assurance process, but also provides prove of a (hopefully) thorough elaborated and utilized quality assurance process.

Although this paper gives only very simplistic, examples, we think that those examples are representative enough to boost you imagination on what is possible. As mentioned before, a key criteria is definitly the speed of the response. If a check after a commit takes more than a couple of minutes, it definitely gets uninteressting (even 1 minute might be too long), so time is the restricting criteria for these checks. Fortunately git operations are very fast, so there are many useful checks possible.

Acknowledgements

This paper has been produced during **Task 4.2: Validation capability at the validation tool level** of the OVERSEE project (FP7-ICT-2009-4, Project ID 248333).

References

[1] *GIT - the fast version control system*, GIT Homepage, *http://git-scm.com/*

[2] *Pro GIT*, 2009, Scott Chacon, *http://progit.org/*

[3] *The Linux Kernel Coding Style*, 2007, Greg Kroah Hartman, *http://www.kernel.org/doc/Documentation/CodingStyle*

[4] *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*, 2004, MISRA Consortium

[5] *5: POSTING PATCHES*, Linux Kernel Documentation , *linux/Documentation/development-process/5.Posting*

[6] *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*, IEC, 1998

[7] *DO-178B - Software Considerations in Airborne Systems and Equipment Certification*, RTCA, December 1, 1992

[8] *Why GIT is better than X*, Scott Chacon,*http://whygitisbetterthanx.com/*

[9] *Stanse - Taking a firm stanse on bugs*, ITI, Faculty of Informatics, Masaryk University, *http://stanse.fi.muni.cz/*