Open Source Methodologies and Mission Critical Software Development

Michael P. Voightmann
Massachusetts Institute of Technology
vmann@mit.edu

Charles P. Coleman
Massachusetts Institute of Technology
ccoleman@mit.edu

Abstract

Reliable, robust software is vital to every successful space mission. This is particularly true with the large risks, complexities and costs involved. Now more than ever, mission software must be well planned, heavily tested and thoroughly debugged. At the same time, priceless expertise is lost every year, and has proven to be difficult to recoup. We believe taking a look at open source software (OSS) methodologies may provide insight into improving the development of critical software.

Three key areas of success in open source software that are directly applicable to mission software, are: (1) broad peer review, (2) code releases early and often, and (3) an excellent training ground. The open source movement has shown that a large group of dispersed developers can create large, reliable systems. The key methods that propel the open source movement can be applied to improving mission critical software.

1. Introduction

There is a huge responsibility and need to produce "ultra-reliable" software. But this goal is in direct conflict with the increasing complexity, size, longevity, and cost of software [26]. This trend doesn't appear to be slowing down. In response to the increasing costs and complexity, organizations are attempting to reuse large amounts of software. This can have disastrous effects, as seen by the Ariane accident. Improvements in the maintenance of critical software could reap great benefits. The ability to quickly (and cheaply) correct and/or enhance code is vital to the current mission as well as the others to come. One approach to this problem may be to borrow ideas from those who do maintenance well, open source software (OSS) developers. OSS is based on the idea that "more heads are better than one" or as Lawrence Lessig put it, "Many tiny brushstrokes of thousands, paints more and more powerfully than the blast of even the most important and powerful papers... [12]."

In this paper, we will briefly discuss the challenge of creating mission critical software. We will then give a description of OSS development. We will look at the possible benefits open source development can provide, as well as point out possible shortcomings. Finally, we will discuss how key open source methodologies can improve mission critical software.

2. Mission Critical Software

Mission critical software must perform flawlessly. A number of approaches have been made to ensure that this software runs smoothly. Key approaches include: adding levels of redundancy, following rigid guidelines, applying formal methods, fault detection, and an extensive amount of testing. Beyond the initial project, software is often reused and thus, it must be ensured to work perfectly in this new setting as well. A study done by Case Associates showed that between 50 to 70 percent of a software engineer's time is spent making changes to mission-critical software. [20] This maintenance work plays a major role in the duties of every software engineer. "Software maintenance is the most time consuming and resource consuming part of the whole software development process." [26] "Today's maintenance efforts can best be described as system-replacement projects, in which organizations migrate or replace existing software functions rather than build them from scratch. Most of the maintenance programmer's time is spent figuring

out legacy code, defining what the current system does, where, and how." [20]

The maintenance problem is much larger for critical systems though. Software maintenance must be much more thorough, much more complete than the normal system, because of the high risks involved. There is much more to lose: with human lives on the line, a much greater amount of money, or particularly valuable information. The bugs that routinely pop up in a web browser or an operating system can not be allowed to happen in a critical system.

Software has often been seen as a system 'component' to be verified individually. Many now seen that "software" cannot be listed as one of the subsystems. Software holds together and interacts with every 'mechanical' component. "Software allows almost unlimited complexity in component interactions and coupling compared to the physical constraints imposed by the mechanical linkages replaced by computers." [13]

Attempting to liken hardware systems to software systems has caused problems for testing. When measuring reliability in hardware, an ultrahigh reliability has historically been seen as a probability of failure on the order of 10-7 to 10-9 for 1- to 10-h missions. But software does not physically fail as hardware does; software is either correct or incorrect with respect to its specification. [6] Software 'failures' that lead to an accident are usually a complex combination of equipment failure, faulty maintenance, instrumentation and control problems, management errors, design errors, and operator errors. [24]

More problems arise when attempting to validate mission critical software.

In particular, software may be highly reliable and correct and still be unsafe when: the software correctly implements its requirements but the specified behavior is unsafe from a system perspective; the requirements do not specify some particular behavior required for the safety of the system; the requirements are incomplete; or the software has unintended (and unsafe) behavior beyond what is specified in the requirements. [13]

Mission critical software must be thoroughly examined. Difficulties arise when trying to determine when enough testing has been done. At least half the resources in software development today are devoted to testing. For safety critical systems, much of this testing is used to build confidence that the software is safe. Unfortunately, the confidence achieved this way is limited. We are able to test only a small fraction of the enormous state space. [13]

The problems that must be faced in creating mission critical software have now been shown. Rather than using the methods currently in use for overcoming these difficulties, we believe a higher level approach may be advantageous. We propose that using ideas from the open source movement could reap benefits at the organizational level. Let's look at what the open source community can add to mission critical software.

3. Open Source Movement

The open source movement has produced robust office suites, operating systems, commercial grade database systems, not too mention the most popular web server, Apache. GNU/Linux is currently the fastest growing operating system [21]. "Open source has played a fundamental role in the development of the Internet by contributing to such remarkable software as TCP/IP, BIND, Sendmail, Linux, and Apache [10]."

3.1 Benefits

There are a large number of benefits to using open source methodologies, which could dramatically improve mission critical software. OSS promotes shared expertise, as anyone can learn from what "specialists" did. Many open products have proven to be reliable, portable, and scalable, attributes every product seeks to attain. The open source approach is often attributed with creating software with "shallow bugs." This may seem obvious; a large number of programmers sifting through code will find more bugs than a small group. Some argue that a small group of specialists can locate bugs better than many non-specialists. The following is an excerpt from "Open Internet Wiretapping":

It is difficult to overstate the value of the kind of widespread review that open source can provide for security-critical systems. Even intense review by small teams of experts often misses small but serious bugs that turn out to have severe security implications. For example, it was only review by the open research community that found several protocol failures in the National Security Agency's "Clipper" key escrow system, in spite of internal reviews by the Agency [5].

This is an example of a key benefit of the open source paradigm, peer review. In every engineering field, peer review is the cornerstone to furthering ideas, yet software source code is rarely reviewed, objectively, on the level OS code is reviewed. The importance of peer review can't be underestimated. As one visible example, Amazon.com began as one of many online booksellers, but their peer book review service helped them to become a dominant online marketplace. The situation was similarly true for the gain in popularity of Google.com. Their search engine ranks websites based off the popularity of the website (how frequently it has been linked to by others). This feedback from a large community is very useful in making decisions.

Another reason OSS has "shallow bugs" is the idea of "releasing early and often." Whenever changes have been made to the project, this new version is released for others to look through and test. This allows for bugs to be found immediately, rather than at the end of the development process. Debugging takes place in parallel with development. This is of great importance, particularly since bugs found early are "cheap" to fix, relative to those found late in the process.

OSS can also have an effect on education. Evaluating other's code as well as having them evaluate one's own leads to great advances in skill. Knowing your code will be seen by many people, particularly peers, tends to lead to better coding habits. Releasing the code for anyone to peruse puts education at the forefront. Open source projects have provided a library full of source code, from real world reliable systems, that any student can learn from.

3.2 Shortcomings

There are a few questions that need to be answered before an organization can jump in and support OSS. Are there any guarantees that the project will ever be completed? That a particular bug will be fixed? Is the software secure? Robust? Safe? In general, can an OS project be mission critical?

Many of these questions can be answered directly by citing examples. Apache, Linux, and Kerberos have shown that secure open source projects can be successful. Developers have created robust, reliable systems that fit the needs of many individuals and organizations. The question of whether the project will be completed can be asked of any project. It would never be wise to blindly use a product. Before you commit to any product, you'd have to take into account how active the developers are in fixing reported bugs, commercial or open source. The main difference is that individuals can contribute to fixing OSS bugs, rather than wait for the next version of a commercial product.

4. Possible improvements

In general, OSS methodologies can improve critical software, largely through peer review, releasing early and often, and education. Even for organizations that don't wish to or cannot give source code to general public, these suggestions can be useful.

1. All code should be viewable, by anyone in the company or at least expanded to those not necessarily connected to the project. Right now, code repositories are common, mainly for a project, group, or a couple of groups, but expanding the availability of the source code is important. It is not at all unlikely that an individual in a different group can give an insight not available in the current group. This also could help greatly curb redundant code.

2. Perhaps an hour a day, developers can read code from other projects, and send contributions to those projects, such as design suggestions, style suggestions, bug reports, or comments in general. Right now, I believe it is rare that developers look at others code, unless they need to fix a part of it. If developers are encouraged to look at others code (and given time to), overall improvements will be seen.

3. A practice that seems to be somewhat common is to have a developer work on a few projects simultaneously. This does help relieve stagnation and molds a "better average programmer." Implementing open source methodologies could take this idea another step forward. A developer can be a key contributor to a small number of projects, as well as an occasional contributor to a larger number of projects. The company and developer will see gains by widening the "average toolset." While coding remains an essentially solitary activity, the really great hacks come from harnessing the attention and brainpower of entire communities. The developer who uses only his or her own brain in a closed project is going to

fall behind the developer who knows how to create an open, evolutionary context in which feedback, exploring the design space, code contributions, bug-spotting, and other improvements come from hundreds (perhaps thousands) of people. [19]

4. Greenspun mentions in Internet Application Workbook that most programmers suffer from profound deficits in: thinking critically about what the computer application should do, writing down a design, writing down an implementation plan, documenting important features/design decisions, clean modular design, exercising good judgement, communicating project status [1]. An open environment can begin to help correct these deficiencies quickly, as bad habits are difficult to conceal when the source code is open.

5. Conclusion

We have seen that maintenance is a crucial part of creating successful mission critical software. It is also the costliest part of the development process. But perhaps maintenance shouldn't be looked at as the expensive phase at the end of the development process. Perhaps maintenance should be thought of as being a vital part of every phase. OSS does this by debugging the source code in parallel with the development process.

"Successful maintenance requires two things: ability to make changes easily and an in depth understanding of the software's structure and behavior [25]." OSS succeeds in both regards, because 1) its projects source code is open to anyone; 2) insight is accepted from experienced programmers as well as first time contributors; 3) source code is released frequently to minimize "bug damage" and 4) open source developers form a competitive community, fostering an excellent learning environment. Taking this all into account, it is easy to see that OSS methodologies can be a great tool to improve mission critical software.

6. References

[1] Andersson, Eve, Phillip Greenspun, Andrew Grummet. *Internet Application Workbook.* 4/2001. http://philip.greenspun.com/internet-application-workbook/

[2] Anderson, Ross. "Security in Open versus Closed Systems – The Dance of Boltzmann, Coase, and Moore." Cambridge University. June, 2002. http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/toulouse.pdf

[3] The Apache Software Foundation. http://www.apache.org/

[4] Augustin, Larry, Dan Bressler, Guy Smith. "Accelerating Software Development Through Collaboration." ICSE 2002.

[5] Bellovin, Steve, Matt Blaze. "Open Internet Wiretapping." 7/19/2000. http://www.crypto.com/papers/opentap.html.

[6] Butler, Ricky W. and George B. Finelli. The infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. IEEE Transactions on Software Engineering. January 1993.

[7] Cioffi, Denis F. "Learning From Hackers." IEEE Spectrum. June 2001.

[8] Hunt, Andy, and Dave Thomas. Software Archaeology. IEEE Software. March/April 2002.

[9] Glass, Robert L. "Frequently Forgotten Fundamental Facts about Software Engineering." IEEE Software. May/June 2001.

[10] Hann, Il-Horn, Jeff Roberts, Sandra Slaughter, Roy Fielding. "Why Do Developers Contribute to Open Source Projects? First Evidence of Economic Incentives." ICSE 2002. http://opensource.ucc.ie/icse2002/HannRobertsSlaughterFielding.pdf

[11] Knight, John. "Critical Task of Writing Dependable Critical Software." IEEE Software. January 1994.

[12] Lessig, Lawrence. http://cyberlaw.stanford.edu/lessig/blog/

[13] Leveson, Nancy G.. "System Safety in Computer-Controlled Automotive Systems." SAE Congress. March, 2000.

[14] Leveson, Nancy G.. *Safeware: System Safety and Computers*. Addison-Wesley. 1995.

[15] Maguire, Steve. *Debugging the Development Process*. Microsoft Press. 1994.

[16] McConnell, Steve. *Code Complete*. Microsoft Press. 1993.

[17] NASA IV&V Facility. http://www.ivv.nasa.gov/faq/index.shtml

[18] Open Source Initiative. http://www.opensource.org/

[19] Raymond, Eric Steven. "The Cathedral and the Bazaar." 2000. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html

[20] Sharon, David. "Meeting the Challenge of Software Maintenance." IEEE Software. January 1996.

[21] Stone, Adam. "Open Source Acceptance Grows." IEEE Software. March/April 2002. p102.

[22] Virginia Tech. "Software Process Models." Aug 98. http://ei.cs.vt.edu/~cs1704/fall.98/notes98/2up/12.SEMod.pdf

[23] Vicente, Kim J. *Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work*. Lawrence Erlbaum Associates. 1999.

[24] Voas, Jeffrey. "Certifying Software for High-Assurance Environments." IEEE Software. July/August 1999.

[25] Wilde, Matthews, Huitt. "Maintaining Object Oriented Software." IEEE Software. January 1993.

[26] Zagal, Jose Pablo, Raul Santelices Ahues, Miguel Nussbaum Voehl. "Maintenance-Oriented Design and Development: A Case Study." IEEE Software. July/August 2002.