Version Control Software for Knowledge Sharing, Innovation and Learning in OS

Maha Shaikh and Tony Cornford
London School of Economics and Political Science, UK.
Email: m.i.shaikh@lse.ac.uk; t.cornford@lse.ac.uk

Abstract. This paper seeks to explore the extent to which version control tools, a common part of the technical infrastructure of much software development and ubiquitous within the open source movement, both represent and facilitate knowledge creation, learning and innovation within open source communities. The paper considers these software actors within open source as both the outcome of innovation and learning, and a means to such ends. Version tools, along with other key technologies, form the infrastructure upon which such communities are built and organized. Their role in knowledge exchange is vital because the ability to produce and observe more than one version of an application is indispensable to dispersed software development.

Introduction

This paper seeks to explore the extent to which version control software, a common part of the technical infrastructure of much software development and ubiquitous within the open source movement, both represent and facilitate knowledge creation, learning and innovation within open source communities. The paper considers these software actors within open source as both the outcome of innovation and learning, and a means to support other such activities within the software development activity.

Version control software has become an almost indispensable part of open source activity, though developers like Linus Torvalds avoided their use for many years (Shaikh and Cornford 2003). Commonly used tools today include Concurrent Versions System [CVS], Subversion, Arch and BITKEEPER [BK]. In a straightforward understanding such tools support knowledge creation, learning and innovation by providing a structured, updateable and interrogateable repository of code versions and patches, together with metadata offering design rationale. By shared access to such systems developers and users of open source software are able, in varying degrees and by varying means, to read, change and distribute code. Beyond such uses version control allows individuals to review past work, to identify particular individuals work and to stamp or tag the changes for easy access. This paper explores such version control software as an actor within open source networks (using the term actor in an ANT perspective), and evaluates its contribution to both individual and community learning and the potential to support innovation as part of the software process and in pursuit of the software product.

Version Control Software[1]

From its beginnings the UNIX community needed a tool to 'manage software revision and release control in a multi-developer, multi-directory, multi-group environment' (Berliner 1990). Distributed development within a diffuse community setting, as in open source, suggests just how integral coordination mechanisms are in geographically distributed development, over and above collocated development (Herbsleb and Grinter 1999). Furthermore, the release strategy of many open source projects support parallel releases, with even-numbered releases relatively stable production versions where the focus is on bug fixing, and odd-numbered releases are experimental and where new features are added and tried out. Such a strategy makes careful control of code versions even more important.

UNIX and its derivatives and developments have had a long history of using version management tools, including the original diff and patch programs, the precursor to most other version tools, with the most popular being RCS [Revision Control System] and SCCS [Source Code Control System] (Koike and Chu 1997). SCCS was one of the first software tools which allowed for a technique of capturing sequenced changes to a module stored as 'deltas'[2] that could be 'strung together in a chain' to show progress in the version (Rochkind 1975). RCS

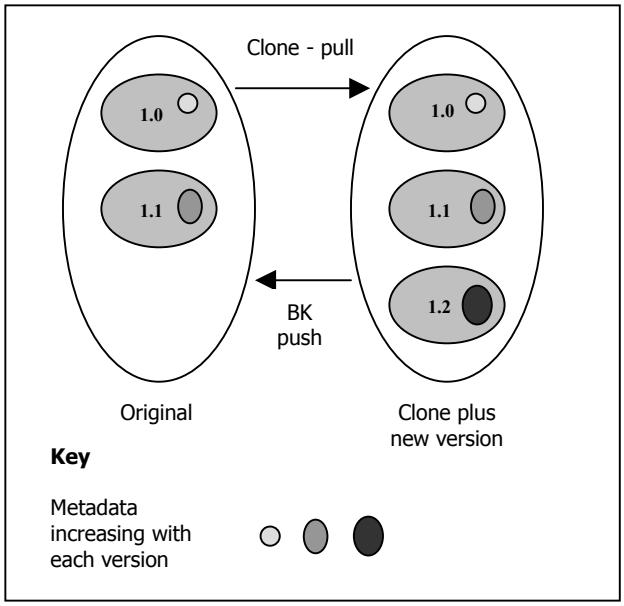
[1] It is common, in the software engineering context to refer to version control tools. However here we are careful to refer to version control software, since we see it as an autonomous element within a heterogeneous network of developers, other technologies, code, networks, markets, artefacts etc. The tool metaphor (if that is what it is) is inappropriate if we see such software as translating and inscribing action within the network.

[2] A delta is an atomic unit of change, for example to a line of code, a variable name etc.

followed the basic principles of SCCS and was designed for 'both production and experimental environments' with automatic identification[3] (Tichy 1985), and supported both forward and reverse deltas (e.g. ability to roll forward by adding changes, or roll back by removing them). CVS [Concurrent Version System] is another such tool, itself an OS product, and one which today, despite being over 20 years old, has wide popularity in OS (Hoek 2000).

BITKEEPER, a more recently developed tool, developed for and used most recently in the Linux development (Shaikh and Cornford 2003), has a refined architecture specifically developed for distributed operation within a hierarchical open source community and originally tailored for the Linux community. To change files a developer must first make a local clone of the original repository; a developer is then free to work on this clone or 'child' independent of the original or 'parent' repository. Changes made to individual files are grouped together into changesets, 'a grouping of one or more deltas to one or more files representing a single logical change' (Henson and Garzik 2002). When the changes made in a child repository are to be merged back to the parent only one merge is needed because, unlike CVS or other software, BK only needs to backtrack to the last merge of changesets in the two repositories. BK's architecture and merging algorithm is also distinctive in that it allows for important meta-data to be stored and generated [see Fig 1].

Figure 1. A BK clone, change commit and push, showing metadata increasing with each version (adapted from Henson and Garzik, 2002)



[3] Identification is the 'stamping' of source and object code of revisions and configurations with unique markers. These markers are akin to serial numbers, telling software maintainers unambiguously which configuration is before them [Tichy, 1985]. CVS calls this function 'tagging' and in BK the changesets themselves perform this role.

Perspectives on Knowledge, Learning and Innovation

We take as our basic perspective in this paper two views of version control software as an actor in the network[4]: 1) as an expression of the learning or innovation that the OS community has engaged in over time- for example in developments such as BITKEEPER, arch or subversion; 2) as serving the acquisition, storage and distribution of knowledge within an open source community as it works to develop software. Both aspects might be seen equally important or significant, but we are concerned here to balance both and to find an appropriate theoretical position to explore them together.

Lanzara and Morner (Lanzara and Morner 2003) provide an ecological/evolutionary perspective on knowledge within open source communities, proposing process that develops by means of 'variation, selection and stabilization', where knowledge is defined as 'the unplanned, evolutionary outcome of the complex interplay of a broad variety of heterogeneous elements' and is classified in three broad types; technical, organizational and institutional. Our initial emphasis in this paper is on what Lanzara and Morner would call technical knowledge, where the focus of analysis is on how new versions of the VC software or application code are generated (variation), accepted (selection) and become part of a new build or new ways of working (stabilization), all of which are potentially mediated through version control software. But in open source such fine distinctions between what technical, organisational (community) and institutional is hard to uphold (just as ANT challenges what is social and what is technical) as the debate over code continues, history is revisited, and the impact of who does what comes into play. In the end the focus of our concern is with a learning collective or community (more specifically, drawing on ANT, a heterogeneous network of interests), which shares characteristics of (or even transcends) being discretely technical, organizational or institutional. Our focus is here is on how version control, as an actor in this network, embodies knowledge and serves to support learning (individual and community), and innovation, and is itself innovated.

Research on innovation in open source has developed strongly over the last few years, but the question of how, or if, learning takes place in open source has not received equal attention. Thus Tuomi (2001; 2002) describes open source in terms of networks of innovation, Franck and Jungwirth (2002) analyse governance issues in innovation and von Hippel and von Krogh (2003) present an open souce derived model of innovation in their private-collective model. Other examples of innovation studies in open source include von Krogh, Spaeth and Lakhani (2003), Francke and von Hippel (2003), as well as innovation oriented sites such as

[4] Of course there is more to the network than just people and version control, and we acknowledge the use of many other participants in the network, including code itself, other web resources such as SourceForge, etc.

Bessen's *Technological Innovation and Intellectual Property* newsletter[5] and Lakhani's web site devoted to user innovation[6].

The perspective on knowledge, innovation and learning we adopt here is derived from Hargadon and Fanelli (2002)[7], proposing a complementary dualistic understanding of knowledge as both a potential for innovation "the generation of new ideas" and a process of learning "converting experience into potential for new actions". This duality is explicitly linked to a structurational argument, in which learning enables and constrains innovation, while innovation provides empirical opportunities that enable and constrain learning. Knowledge is thus seen as both latent (learning) and empirical (innovation), but this is not suggested as a binary categorization, one or the other, but as a duality in which one creates or shapes the other. Latent knowledge suggests the capacity to generate novel organizational artefacts, while empirical knowledge resides in an organization's actions or behaviours, or more precisely in the artefacts created like technologies, databases, and operating procedures – or in this case code and software actors. Hargadon and Fanelli argue that this complementary approach 'shifts the focus away from either learning or innovation and towards the interaction between them'.

Knowledge Sharing in OS through Version Tools

Version tools serve as an actor that filters and structures communication, information updating, collaboration and knowledge sharing. Knowledge sharing is key because such an actor provides a mediated organizational memory, containing metadata (rationales for code changes, identification of changes etc.) and an ability to generate versions of software at various stages of development and that can be reviewed and traced through time.

Hargadon and Fanelli's notion of empirical knowledge stresses learning and action. Actions here imply physical and social artefacts and include the deltas and changesets. Version tools and the architecture they rely on form part of the social and physical infrastructure. Combined with the Internet, chat groups, email and mailing list archives version tools make open source development a reality. Open source communities require an open environment and open tools for development [though BK is not an 'open' tool]. Source code is open for all to see, read and modify thus it allows developers to 'learn' from the work of other developers. This in turn implies that any tool that is used in open source development should

---

5 http://www.researchoninnovation.org/tiip/index.htm

6 http://userinnovation.mit.edu/

7 Their paper is specifically oriented to knowledge in organisations. In this brief paper we equate organisation with community, while recognising that the shift between the two concepts requires careful consideration.

facilitate this openness. The very purpose of version tools is to help coordinate development in such a way so as to allow the developers to view and modify the work done before.

Organizational Memory

Organizational memory is one way of describing how 'organizations build stores of knowledge' (Hargadon and Fanelli, 2002) and version 'tools are repositories of knowledge about a project' (Cubranic, Holmes et al. 2003). Version software used by developers, be it CVS, BK or Subversion, is itself constantly undergoing adaptation and improvement. Such tools have insinuated themselves into dispersed development and become indispensable [an obligatory passage point] to open source. This is a way that version software [making itself integral] ensures that it will attract more developer attention and work. Such a process is quite reflexive because any 'learning' on version software development can also be steered into other open source software development.

VGER, the centralized repository of CVS holds all the versions and deltas of the Linux kernel. Scott Hissam[8] believes that if you can make a link between tools and source code then you have a true organizational memory. Larry Augustin of SourceForge.net [at the same workshop] added that CVS does act as an organizational memory 'but it takes more than source code management, you need archives, repository, IRC chats, which all together become organizational memory'. He then continued to make a link to his web knowledge portal, SourceForge and claimed that 'having an organizational memory is one of the top three selling points of SourceForge'.

Source Code and Metadata

Source code is the output of past learning but in turn it becomes the inspiration for further learning and innovation. It is a central actor that has the power to make [inscribe] developers into adding and improving it. When a patch or bug fix with its' source code is released it is scrutinized by a number of co-developers who either work to improve it or add features to it, thus source code is also a tool for 'knowing and organizing' (Lanzara and Morner 2003). Source code in every version or small patch is reviewed by many eyes (Raymond 1998)[9] but this is only possible because the source is open, and in turn what makes this testing more efficient [and some would say even possible] is the use of version tools and their ability to allow simultaneous dispersed software development.

Source code is not always understandable on its own. It requires at least some form of explanation and this function is performed by the metadata. Metadata is

---

[8] This is a part of the discussion from the Open Source Workshop at Portland, Oregon in May 2003.

[9] This is what Raymond (1998) termed Linus's Law of 'given enough eye-balls, every bug looks shallow'.

the notes and comments written by developers which make their patch or bug-fix understandable. Metadata consists of the name of the author, creation time and any or all the checkin comments (Cubranic, Holmes et al. 2003). BITKEEPER supports developers adding metadata, and indeed this metadata is considered to be vitally important as one can see from the dispute between BITKEEPER creators and some CVS users (Shaikh and Cornford, 2003) over McVoy not converting all the metadata of BK into CVS compatible form. And McVoy admits that 'information about who did what, and maybe why they did it, is recorded and is useful for *learning* the source base, tracking down bugs, etc.' (McVoy 2003).

Deltas, Changesets and Release Strategy

The most recent version of the trunk [main section of code] is saved in complete form [rectangle 2.2 in Fig 1] and all past revisions are saved as reverse deltas [triangles 1.1, 1.2 etc in Fig 1]. The reason why forward deltas are helpful is because they allow for branches of the version to be made. In order to re-create a revision on a branch a developer must first extract the latest revision on the trunk, apply reverse deltas until the fork revision for the branch is obtained and finally to apply forward deltas until the required branch revision is arrived at (Tichy 1985). BK's changesets operate as bunches of deltas so only single pulls or pushes need to be made (Henson and Garzik 2002). These various deltas allow developers the ability to read not just the various patches but also the metadata accompanying each patch. Developers build on the work of others where in certain cases innovation can be said to occur as something novel is created but in support of Hargadon and Fanelli's complementary approach 'I don't believe in truely new work. Everything is a new light on a pile of existing technology. Whats BK but a collision between graph theory, CVS and distributed resolution stuff. BK builds on that knowledge' (Cox 2003).

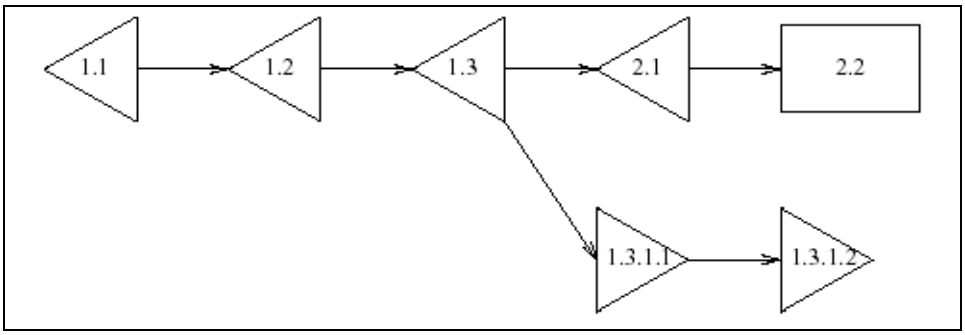


Figure 2. Reverse and Forward Deltas (Tichy, 1985).

Conclusion

Learning and innovation are related concepts, 'learning fuels innovation' (Cox 2003). This paper made an effort to delineate some of the features of version control tools which facilitate knowledge creation and sharing in open source because,

> "The concept of *organizational* knowledge can be understood only as the result of an ongoing, circular interaction between individually held latent knowledge and the knowledge manifest in the surrounding environments. It is only through this interaction that knowledge emerges as a social (and thus organizational) phenomenon." (Hargadon and Fanelli 2002)

This is true of learning and innovation in open source because as this paper has attempted to show learning occurs through a combination of human and technical actors interacting, where version software becomes both a vehicle and a product of learning. There is however a need to consider how such tools;

- Allow for unlearning (as in the case of the Linux kernel developers moving from CVS to BK and then partially back again with the BK to CVS gateway facility,
- Cause loss of learning, especially in the case of the BK to CVS gateway where some precious metadata is alleged to have been lost in the translation of software tools and,
- Influence learning in a certain direction so as to translate the tool's 'needs' and control over the human actors.

References

Berliner, B. (1990). CVS II: Parallelizing software development. 1990 Winter USENIX Conference, Washington, D.C.

Cox, A. (2003). Re: Why DRM exists [was Re: Flame Linus to a crisp!], University of Indiana. 2003.

Cox, A. (2003). Re: Why DRM exists [was Re: Flame Linus to a crisp!] - part 2, University of Indiana. 2003.

Cubranic, D., R. Holmes, et al. (2003). Tools for light-weight knowledge sharing in open-source software development. 25th International Conference on Software Engineering - Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering, Portland, Oregon.

Franck, E. and C. Jungwirth (2002). Reconciling investors and donators - The governance structure of open source, Working Paper Series, Chair of Strategic Management and Business Policy, University of Zurich. 2002.

Franke, N. and E. v. Hippel (2003). "Satisfying heterogeneous user needs via innovation toolkits: the case of Apache security software." Research Policy 32(7): 1199-1215.

Hargadon, A. and A. Fanelli (2002). "Action and Possibility: Reconciling Dual Perspectives of Knowledge in Organizations." Organization Science 13(3): 290-302.

Henson, V. and J. Garzik (2002). BitKeeper for Kernel Developers. Ottawa Linux Symposium, Ottawa, Ontario Canada.

Herbsleb, J. D. and R. E. Grinter (1999). Splitting the organization and integrating the code: Conway's law revisited. 21st International Conference on Software Engineering (ICSE 99), Los Angeles.

Hoek, A. v. d. (2000). Configuration Management and Open Source Projects. 3rd International Workshop on Software Engineering over the Internet.

Hippel, E. v. and G. v. Krogh (2003). "Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science." Organization Science 14(2): 209-223.

Koike, H. and H.-Chu Chu (1997). VRCS: Integrating Version Control and Module Management uisng Interactive Three-Dimensional Graphics. IEEE Symposium on Visual Languages (VL'97).

Krogh, G. v., S. Spaeth, et al. (2003). "Community, joining, and specialization in open source software innovation: a case study." Research Policy 32(7): 1217-1241.

Lanzara, G. F. and M. Morner (2003). The Knowledge Ecology of Open Source Software Projects. European Group of Organizational Studies (EGOS Colloquium), Copenhagen.

McVoy, L. (2003). Re: BK->CVS, kernel.bkbits.net, University of Indiana. 2003.

Raymond, E. S. (1998). "The Cathedral and the Bazaar." FirstMonday: A Peer Reviewed Journal on the Internet **3**(3): 1-24.

Rochkind, M. (1975). "The Source Code Control System." IEEE Transactions on Software Engineering 1(4): 364-370.

Shaikh, M. and T. Cornford (2003). Version Management Tools: CVS to BK in the Linux Kernel. 25th International Conference on Software Engineering - Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering, Portland, Oregon.

Tichy, W. F. (1985). "RCS—A system for version control." Software - Practice and Experience 15(7): 637–654.

Tuomi, I. (2001). "Internet, Innovation, and Open Source: Actors in the Network." First Monday 6(1).

Tuomi, I. (2002). Networks of Innovation: Change and Meaning in the Age of the Internet. Oxford, Oxford University Press.