Constitutional Dynamics of the Open Source Software Development

Jukka Kaisla
Department of Industrial Economics and Strategy
Copenhagen Business School
Howitzvej 60, 2000 Frederiksberg, Denmark
jukka.kaisla@get2net.dk

May 2001

Introduction

This paper maintains that the nature of conventions and social contracts among the software developing community have been decisive in the open source software development. It will be explained that purposeful design has played a minor role in the overall success of open source projects, such as Linux operating system.

In addition to conventions and social contracts, the paper will examine the importance of three other factors: (1) *objective knowledge* aspect of the shared source code, (2) *modularity* of software, and (3) *selection* of software by the project management. The conclusion will be that the extent to which these elements are of designed origin, their design is not ultimately critical to open source software development.

The open source software model is based on the freedom to use, copy, modify and redistribute software. The term *open source* means that the source code needed to modify software is provided, and that the users/developers have the right not only to use, but also to modify and distribute modified versions. The starting point is that nobody is permitted to pronounce an exclusive property right to open source software. The proprietary model with which the open source model is convenient to be compared is based on a more conventional idea of copyright. The developer/distributor reserves all rights to copy, modify and distribute while users have only the right to use the software.

The sketch of the complex and interdependent model is as follows. The elements of the model are examined in terms of their degree of intentional design vs. unintended impact, as well as in relation to their degree of importance or necessity to the process. The analysis will begin by looking at general conventions of fairness among the software-developing community. These conventions are unintended from the open source software development point of view. The conventions of fairness give rise to specific conventions of 'property' in open source development. Drawing upon these conventions, the central players in open source development designed a social contract to maintain the beneficial pattern of cooperation among developers.

Open source software itself brings important elements to the model as well. Three elements are considered here: technological modularity which is viewed here as comprising both intentional and unintended elements, the objective knowledge aspect of source code as an enhancement to communication (an unintended element), and the selection process of software improvements which is an intended element in the model. All these elements together give rise to *interests* and *capabilities* of the members to participate in the development of open source software. Genuine uncertainty of the overall interplay between these elements was implied by Linus Torvalds, the founder of Linux, the prominent open source operating system: 'Only afterwards have we started thinking about what went right in the process' (Wow, 1 June 2000), and 'Linux emerged by coincidence' (Wow, 28 Nov 2001).

The paper is organised as follows. I begin by discussing the background of open source software development. The second following section will examine open source conventions and social contracts, together with some central reasons for their emergence and

enforcement. In the third section, I examine the modularity, objective knowledge and selection related aspects of open source development. The fourth section will examine the interplay among the spontaneous and the purposefully designed elements of the model. A central issue for the future of the open source paradigm appears to be the question about how the social contract will develop.

Open source software – aims and rationales

Open source software development is looked upon today with increasing astonishment. From the rational maximisation perspective, it should not exist or at least not spread as fast as it does. Acquiring, developing and distributing open software is free of charge. The developers do not receive the right to own their contribution and are required to provide access for anybody to obtain their contribution. Access to and distribution of software is facilitated by modern technology, especially by the Internet and e-mail news groups.

The beginning of the open source movement, in the early 1980s, was a conscious attempt to continue the software-sharing conventions of the software developers' community. Sharing and exchanging software freely among the developers was the convention before; in the early 1980s prominent university laboratories and companies started using nondisclosure agreements to prevent the distribution of free copies (Stallman 1999). The software-sharing convention at that time was rational from the developer's point of view, as income streams were not connected to choices whether or not to distribute copies and modifications. The game was reciprocal where everyone gained by helping and receiving help from others. But the game can go on only as long as copyrights and licenses do not prevent it — and they started to do precisely that.

There were many reasonable reasons for the increasing use of copyright and restrictive licenses in the 1980s. Without going too deep into that line of discussion, one can hypothesise that the change from huge central computers toward personal computers was an important factor in the development. The rise of proprietary software made some members of the software developers' community uncomfortable. The question was not so much about whether it was morally correct for somebody to make money out of developing and selling useful software. It was perhaps more about how they perceived software in general. They viewed software as a general means to help people – very much like language. Nobody would like to see our common language being closed up by someone who would then have the sole right to modify and distribute it.

The open source movement arose as a countermovement to the proprietary model. In order to be able to resist the increasing dissemination of proprietary software, open source developers needed to create their own operating system, and the 'GNU' project was born (Stallman 1999). The GNU project was built upon a set of principles that can be viewed as the *social contract* of open source movement. The terms of this social contract, called *Copyleft*, were later on considered too extreme by developers who saw that in order to attract the attention of

business people, they need to alleviate/omit some terms to facilitate the combination of the open source and the proprietary models. This process appears to be increasingly in the core of open source software development today.

A distinctive organisational aspect of open source software development is that there are no predefined boundaries to an open source software organisation. Membership in a project is based on self-selection where those developers who feel capable of contribution do. An open source software project uses software development capabilities throughout the world. Suggested improvements and modifications are then reviewed by a central agency, the project management, which has the right to select between beneficial and less beneficial suggestions. The Internet functions as a prominent means of coordination and communication among developers.

A central distinction between open source and proprietary approaches in software development is that the proprietary approach allows the developers to collect rent from the secret bits of their software, while on the other hand, it forecloses the possibility of truly independent peer review. The open source approach sets up conditions for independent peer review, but precludes the extraction of rent from the secret bits (Raymond 1999).

A central issue that open source software developers need to tackle is the special structure of rights and responsibilities. The rejection of the conventional property rights structure complicates the accountability of each developer. As the social contract does not encourage demarcation of various rights among developers, conventions emerge to remedy the situation. Open source development benefited from building upon conventions that had been developed in software-developers' communities earlier. The open source conventions need not be discovered in the genuine sense because for those who shared the earlier cooperative behavioural pattern, they are rather obvious remedies to the problems that would predictably arise in their absence. Thus, the existence of multilateral reciprocity among software developers influence their procedural interests to continue cooperating even if the property rights in the software world were changing toward the proprietary model.

Open source social contracts and conventions

Open source software development is based on a peculiar pattern of rights and responsibilities. This section will analyse the terms of the open source social contract, which was intentionally designed to preserve open development, and the conventions that arose to frame this development. The social contract prevents anyone from pronouncing exclusive property rights to open source software, whereas central open source conventions function precisely to define particular property rights. There is an interesting interplay between the deliberate aim of the social contract and conventions that define boundaries between acceptable and unacceptable behaviour.

To complicate things, there is an additional set of principles, the Open Source Definition (OSD, see references), which was designed to

provide more closure than the social contract, the Copyleft. A number of licenses have emerged based upon the OSD. The development of those licenses shows a tendency away from the original social contract towards a hybrid version of open source and proprietary principles.

Copyleft and GPL — the original social contract

The aim of the open source movement was to counterbalance the increasingly proprietary world of software development. In order to secure that open source software, after having left the hands of the original developer, remains open source, a legally binding set of rules needed to be established. The solution was found in the combination of copyrighting and licensing. Copyright resolved a problem that, e.g., public domain software suffered from. Public domain software is free in the extreme sense that anyone is free to take a copy of such software, pronounce it as her own, change the author (or any other) information, and start selling it under whatever license she wishes.

The open source people were knowledgeable of the risks that complete freedom might bring about (such as converting open source development into closed source), so they chose to copyright their software, and to provide the General Public License (GPL, see references) based upon the principles of Copyleft to go with it. Copyleft uses copyright law but functions as the mirror image of the conventional use of copyright. The central idea of Copyleft is to give everyone permission to run a programme, to copy, redistribute, modify, and distribute modified versions — but not the permission to add restrictions to the license. It is important to notice that the freedom Copyleft provides does not have anything to do with price. Anyone is free to charge anything one wishes from (re)distribution — as long as the same opportunity is open to anyone else as well.

The central aim of Copyleft being the prevention of open source software from becoming converted into closed source, some important, although unintended, implications follow. A central license design problem is that the designer must not only consider various activities a licensee is prevented from doing, but she must also imagine various ways a licensee could circumvent any of the license terms. The aim of the GPL license is not to prevent people from distributing GPLed software together with closed source software using the same medium (such as a CD-rom). To be sure, the open source principle would have nothing against combining open and closed source software into an aggregate programme, if it were possible to demarcate where one license starts and another ends. This is, however, technically next to impossible and would provide ample opportunities for the more restrictive license to encompass the less restrictive, the end result being that the whole programme would be interpreted through the more restrictive license.

For this reason, GPL contains a term that permits distribution only as 'independent and separate works' with software based on a license more restrictive than the GPL. An attempt to combine GPLed software with another based on a more restrictive license is legitimate only if the resulting whole becomes GPLed. This is why GPL is considered viral or contagious. But we need to recognise the motivation behind this viral nature. The clause is there to protect the less restrictive license from

being interpreted through the more restrictive, in other words, it prevents GPLed software from being hijacked by closed source software. I will turn to this point below when the more relaxed Open Source Definition is discussed.

Open Source Definition (OSD) - the revised social contract

Open Source Definition (OSD, see references) is a bill of rights for the recipient of open source software. It functions as a certificate that ensures that licenses accepted by OSD meet the required criteria and can thus be defined as open source licenses (Perens 1999). OSD grew from a certain degree of discomfort with the demand of symmetry and reciprocity in Copyleft and GPL. The developers of OSD wanted to better be able to connect with the closed source world and still ensure that open source software remains open source. Here are the OSD terms and a short analysis on their function:

1. *Free redistribution:* a license based on OSD may not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programmes from several different sources. The license may not require a royalty or other fee for such a sale. The rationale behind this clause is to promote free redistribution by eliminating incentives for extracting rents on others' work. This clause has the effect of retaining the game cooperative.
2. *The source code must be included*. This clause enhances the development of open source software as modifications are often impossible without having access to the source code.
3. *Derived works*: a license must allow modifications and derived works to the original software, and must allow them to be distributed under the same terms as the license of the original software. For rapid development of software, people need to be able to experiment with and redistribute modifications. This clause has an interesting implication, as it does not require any producer of a derived work to use the same license terms as the original, it only provides an option to do so.
4. *Integrity of the author's source code*: a license must explicitly permit distribution of software built from modified source code and it may require derived works to carry a different name or version number from the original software. This clause enhances reputation building among developers. People need to know who is responsible for particular modifications. The term also facilitates the distinction between official and unofficial changes to software.
5. *No discrimination against persons or groups*. This clause is based on the recognition of the Hayekian problem of dispersed knowledge. Promoting diversity of people and groups equally eligible to contribute is viewed beneficial because

we do not know beforehand who will discover something valuable.

6. *No discrimination against fields of endeavour*: for example, a license may not restrict software from being used in a business. This clause encourages commercial use of open source software.
7. *Distribution of license*: rights attached to a programme must apply to all to whom the programme is redistributed without the need for execution of an additional license by those parties. This clause prevents any attempt to indirectly close up software, such as requiring a non-disclosure agreement.
8. *The license must not be specific to a product*: rights attached to a programme must not depend on the programme's being part of a particular software distribution. This clause facilitates extracting open source software from any distribution, and preserving the extracted software with the same rights as those that are granted in conjunction with the original software distribution.
9. *The license must not contaminate other software*: a license must not place restrictions on other software that is distributed along with the licensed software. This clause facilitates the *distribution* of open source software along with proprietary software, but at the same time it restricts *combining* open source software with proprietary software under the license of the latter. So, any combined work needs to be distributed under OSD.

The third clause on derivative works contradicts the terms of Copyleft and the GPL insofar as more restrictive terms can be introduced to the modification. What this clause does is that it opens up the possibility to privatise modifications and charge money from their use. The OSD conformant BSD license (see references) provides precisely this. However, OSD restricts charging money from the initial license only, so the holder of the initial license is restricted from charging anything from the subsequent redistributions. This creates a tendency for the price of BSD licensed software to approach zero, but it also permits converting a derivative work on BSD software into closed source.

The critical point in preserving open source development open also in the future appears to be the modifiability of license terms. The GPL license terms themselves are outside the rights that the license provides, that is, the GPL defines rights to software which does not include the license itself. By this it prevents any attempt to modify the license terms and can thus guarantee that software which is initially distributed under GPL also remains under it, irrespective of how much it will be modified during the development. The modifiability of the BSD license terms does not provide any guarantee of the future development of open source and is thus vulnerable for rent seeking.

Open source conventions

Open source conventions are based on fairness, non-discrimination and equal treatment of all parties. While most open source developers do not object to others profiting from their contribution, most also demand that no party be in an exclusive position to extract profits. A developer is willing to let someone else to profit by selling her software or patches, but only as long as the developer herself could also potentially do so (consistent with both Copyleft and OSD).

Developers have observed that licenses that include restrictions on and fees for commercial use have serious chilling effects. Restrictions on use, sale, modification, or distribution inflict cost of conformance tracking and, as the number of packages people deal with rises, uncertainty and potential legal risk increases. This outcome is considered harmful, and there is therefore social pressure to keep licenses simple and free of restrictions. Despite this convention, new variants of more restrictive licenses have been developed (such as the BSD). A potential source for this development are aspirations to benefit from the available open source software together with the positive value of the open source label, and at the same time to gain a monopoly position through exclusive rights to software.

A central function of open source conventions has to do with preserving the peer-review culture based on multilateral reciprocity. License restrictions designed to protect intellectual property or capture direct sale value often have the effect of making it legally impossible to fork[1] the project. While forking is considered a last resort, it is considered critically important that that last resort be present as protection against a maintainer's incompetence or defection (Raymond 1999).

The open source social contracts (both the Copyleft and the OSD) permit that anyone can hack anything. Nothing prevents half a dozen different people from taking any given open source product, duplicating the sources, running off with them in different evolutionary directions, all claiming to be 'The' product. In practice, however, such forking almost never happens. Splits in major projects have been rare, and always accompanied by re-labelling and a large volume of public self-justification. The open source movement has an elaborate but largely spontaneous set of ownership conventions. These conventions regulate who can modify software, the circumstances under which it can be modified, and who has the right to redistribute modified versions back to the community (Raymond 1998):

- There is strong social pressure against forking projects. Forking does not happen except under special conditions, with much public self-justification, and with a renaming.
- Distributing changes to a project without the cooperation of the moderators is disapproved.

[1] Forking means to take any given open source product, to duplicate the sources, and to develop them in different evolutionary directions.

- Removing a developer's name from a project history, credits or maintainer list is not permitted without the person's explicit consent.

What does 'ownership' mean when property is infinitely reduplicable, highly malleable, and there are no explicit coercive power relationships in the surrounding culture? The owner(s) of an open source software project are those who have the exclusive right, recognised by the community at large, to redistribute modified versions. According to the standard open source licenses, all parties are equals in the evolutionary game. But in practice there is a well-recognised distinction between 'official' patches, approved and integrated into the evolving software by the publicly recognised maintainers, and 'rogue' patches by third parties. Rogue patches are unusual, and generally not trusted (Raymond 1998).

Conventions encourage people to modify software for personal use when necessary. Conventions are also rather indifferent to activities of redistributing modified versions within a closed user or development group. It is only when modifications are posted to the open source community in general, to compete with the original, that ownership becomes an issue.

There are, in general, three ways to acquire ownership of an open source project. One is to set up a project. When a project has only had one maintainer since the beginning and the maintainer is still active, convention does not even permit a question as to who owns the project. The second way is to have ownership of a project to be transferred by the previous owner. There is a clear convention that project owners have a duty to pass projects on to competent successors when they are no longer willing or able to invest needed time in development or maintenance work. The third way to acquire ownership of a project is to observe that it needs work and the owner has disappeared or lost interest. The responsibility of the acquirer is to make an effort to find the previous owner. If the previous owner cannot be found, then the acquirer may announce in a relevant place (such as a Usenet newsgroup dedicated to the application area) that the project appears to be orphaned, and that she is considering taking responsibility for it. Convention demands that the acquirer allow some time to pass after the announcement. In this interval, if someone else announces that they have been actually working on the project, their claim exceeds the newcomers. It is considered good form to give public notice of the intentions more than once.

These features suggest that the conventions are not accidental, although they may be spontaneous responses to the social contracts that do not clearly define property rights among the developers; spontaneous in the sense that such conventions are increasingly conformed to within a group facing such a social contract. Later on in this paper the open source conventions are examined against the background of an ancient body of natural law.

Objective knowledge, modularity, and selection

In this section, three further components of the model are introduced: (1) objective knowledge, (2) technological modularity, and (3) project management in open source software development. The central aspect in the communication structure considered here is unintended, it has to do with the nature of software *per se*. The technological modularity demonstrates both intentional and unintended aspects of open source development, and the same goes for project management.

Communication and objective knowledge

At first glance, concepts like *informal networks* or *communities of practice* seem to illustrate well what is going on in open source software organisations. A well functioning organisation needs appropriate means for communication and knowledge sharing among its members. Whenever informal networks appear, they tend to generate their own norms and conventions to facilitate communication, thus constituting communities of practice (Crane 1972, Lave and Wenger 1991). This happens both within and across organisations.

The development of structure in a community of practice depends on the overall *size* of the community and on the *diversity* of skills available. Collaborative performance enhancement depends not only on these two factors but also on the *rates* at which the members produce results that are beneficial for the whole community (Huberman and Hogg 1995, 74). Huberman and Hogg advocate an idea of a natural limit, or bandwidth, to the number of people an individual member can interact with in a network. This limit ranges from types of situations where the members can interact with everybody very rarely to types where a limited number of members interact very often.

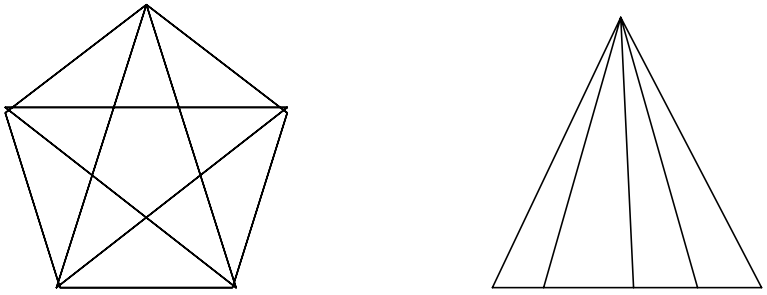
Open source software projects can be analysed, however, through an alternative model of communication, which is less limited by the natural bandwidth effect. It differs from the basic network model in that the members need not interact directly with each other. There is a component that facilitates the flow of knowledge beyond what the members could attain when interacting directly with each other. This component is the *objective knowledge* inherent in the software itself (cf. Popper 1972). What makes knowledge within an open source project so unique is the *source code* that is always provided together with the binary version.

Consider two software developers who try to communicate some functionality problems in a closed source programme, say Microsoft Word. Neither of them has the access to the source code as they do not work for Microsoft. When they discuss the problem they need to continuously interpret and reinterpret what the other party is saying and meaning because they lack an exact language that would require little or no interpretation. The source code provides precisely that function in two distinguishable ways: (1) by being an exact *language*, and (2) by being *objective knowledge* by which developers can coordinate (through trial and error) their subjective knowledge. Language can be viewed as part of the

body of objective knowledge, but here language is discussed as the meaning of a *means* of communication, separate from the knowledge content of any particular sentence. This distinction can be found in, e.g., computer languages that can function simultaneously as a shared language among software developers (coordinator of meanings) and as carrying out objectively existing functions (a piece of code has an effect in software disregarding how it is interpreted).

To see the difference between the network model of communication and the one suggested here consider the following figure 1:

Figure 1: Communication models



All-channel interaction Communication through object

Here we have two communication models among five members. In the first model, the members communicate directly with each other while in the second model an object (such as software) functions as an objective entity to which each member relates. A core difference between these models is that in the first alternative the members need to find out who knows what at each instance, whereas in the second model the objective entity coordinates the type of knowledge that is needed at each instance. In the first model, communication among the members is limited by their abilities (including the costs) of maintaining versatile connections (the bandwidth) whereas in the second model, only those members who at a particular instance perceive being able to add value to the development process do so. Open source software is rather an extreme case as it functions as an exact language and as objective knowledge at the same time.

Cusumano (1997, 9) suggests that small teams conducting complex tasks are more effective than large ones because it is easier to have good communication and consistency of ideas among team members. Two issues are of interest here. First, the question of what is meant by good communication and consistency of ideas. Second, the issue of knowing in advance who will know something valuable in the future.

Good communication is assumed here to be directed at a target (such as software which Cusumano's article deals with). Good communication may mean that things that are understandable by the majority or all members are communicated. Frictions in communication may be due to some members being smarter than the rest, or less smart (among many other reasons). Consistency of ideas is linked with good communication. What the members perceive as good communication can be the result of the consistency of their ideas. It is, however, not

clear to what extent consistency of ideas works well as a primary criterion when complex systems are being developed. A novel idea may be in conflict with the established pattern of consistency, and thus become rejected before it is assessed to its full potential. A small team may work well in resolving *conflicting interests* among the group members, but the smaller the group the less versatile ideas it can produce.

This links us to the Hayekian knowledge problem, i.e., to our ignorance of who may be in the best position in the future to resolve particular problems. If the group members are defined from the beginning, then only those discoveries can be made that are perceived by the members. If then consistency works as the moderator of ideas, only those discoveries are recognised that are consistent with the patterns that are already established. Discoveries become thus limited in two steps: first, by group size, and second, by the consistency requirement.

Technological modularity

Cusumano (1997) describes how Microsoft makes large teams work like small teams. The core strategy is to break both the organisation and the products into subunits to facilitate coordination among the members and product components. The keyword is *modularisation*, both at the organisational and the product levels.

Modularity refers to a general set of principles for managing complexity. Modularity is attained by breaking up a complex system into discrete subunits which can communicate with each other only through standardised interfaces within standardised architecture (Langlois 2000, 1). By doing so a development team can prevent the design process from becoming excessively complex at many levels at the same time. The keyword in modularisation is thus *standardisation* of the critical interfaces that subunits interact with. The degree of modularity in a system can be assessed by examining to what extent small changes in one part of a system lead to unpredictable outcomes in other parts of the system. If a system is decomposed into modules, then changes in one module do not affect others. What modularity does is it breaks the interdependency among the subunits as each module interacts solely with the common interface.

Modularity within organisations can be divided into different types: modularity of the organisation itself, modularity of the products, and finally, modularity of property rights within the organisation. Langlois (2000) suggests, contrary to Sanchez and Mahoney (1996), that technological modularity does not necessarily presuppose organisational modularity. Indeed, there seems to be no compelling reason to assume that product modularity necessarily leads to organisational modularity.

Cusumano (1997) describes how Microsoft applies both organisational and technological modularity to coordinate and stabilise software development. Open source software, like Linux-derived operating systems, demonstrate a high degree of technological modularity but a lower degree of organisational modularity. According to Cusumano (1997) in large development projects in Microsoft, 'many team members create many components or features that are interdependent but difficult to define accurately in the early stages of the development cycle' (p. 10). And also that they need to continuously

'synchronize what people are doing as individuals and as members of teams working in parallel on different features, and periodically stabilize the evolving product features in increments as a project proceeds' (p. 11). The strategy is to continuously iterate among several designs, builds, and testing while developing a product (ibid.). All this seems to indicate that, contrary to Cusumano's view on modularisation in Microsoft, their product development is in fact non-modular. Modularity would prevent interdependency problems and activities resulting from these: continuous synchronisation and iteration as projects evolve.

The object oriented model of communication shown above illustrates open source software development. Consider the object being decomposed into modules each interacting with a standardised interface. In open source software development, the team responsible for developing a particular feature is not defined in the beginning of the project. Instead, the team itself evolves according to the capabilities of individual members to resolve particular problems that arise during the development. Communication among developers is facilitated through interfaces and is carried out in specific arenas (e.g., discussion groups on the Internet) for communicating particular issues. The organisation itself is non-modular in the sense that there is no team exclusively defined to various development projects. If a developer identifies the ability to contribute to a project at a specific instance, she can freely do so. A central benefit from not limiting the development team is that more discoveries and innovations arise during the development process. Another beneficial aspect of keeping the development team open is that we do not know in advance which developer might resolve a problem arising from the previous round of improvements. The development of open source software show a dramatically higher speed of improvements and debugging than what is achieved within the closed source development (e.g., stability and speed of development of Linux vs. Microsoft Windows).

Selection

Open source software organisations are open, non-hierarchical systems. Project management can be distinguished from other members, however. The management normally consists of the property rights owners (defined by convention). The development of the open source operating system Linux has involved a myriad of extensions and improvements along the years, and yet its initial developer, Linus Torvalds, holds the position to unilaterally select among potential improvements. This suggests two unrelated issues: first, the strength of the property right conventions in open source development, and second, a conjecture about the respective importance of variation and selection in software development.

As time passes, the weight of other developers' contributions to any given project normally increases. As in the case of Linux, the initial developer may limit his tasks to almost solely selecting incoming suggestions. The principle of *prominence* may play a central role in sustaining the property rights convention. As years pass and thousands upon thousands of developers have contributed to the development, the

only prominent person who stands out is the one who has held the right to select among trials.

This leads us to an interesting suggestion: prominence does not necessarily arise from the critical nature of the task, but perhaps from a simpler fact that the person who selects stands out because of her role as the initiator. The chain of thought goes something like this: empirical findings show that open source software demonstrates specific strengths over closed source alternatives. These have to do with the speed of improvement and bug fixing, reliability and stability, among other things. This being a general pattern it is hardly likely that open source project managers just happen to be superior in selecting good suggestions from bad ones. Rather, a potential explanation would be that selection is not the central problem, whereas creating variation is. An experienced developer can perhaps easily see what suggestions are worth looking into. And then, technological modularity enhances testing and assessing new variants. Creating variation is precisely what open source software development is superior in. The number of suggestions (variation) to any open source project of some interest exceeds what a coherent closed source development team could ever come up with.

This links us back to the nature of prominence in the open source property right convention. Insofar as selection is not the critical issue, but the creation of variation is, important contributions should have some role in the property right structure. The result would be that open source software would be 'owned' by many, instead of by few. This would, however, be dysfunctional from the project management point of view. Consider suggestions for improvements being voted on in discussion groups. The dysfunctionality of voting assumes of course that software developed by voting would not be any better than another developed by the single selector model. The fact that voting is not generally used promotes the argument that selecting is not the central problem.

This section has suggested three central features to the open source software model: (1) acknowledgement of the dispersed nature of knowledge and of the problem of stimulating the growth of knowledge, (2) communication through an objective entity that functions as a communication interface among the members, and (3) technological modularity of the software.

Dynamics of the interplay

As explained earlier in this paper, open source social contracts (Copyleft and OSD) and conventions work in opposite directions. The social contracts facilitate open development by preventing exclusive property rights while conventions define property rights among the members. It is, however, important to notice that social contracts and conventions have a common origin, namely conventions. A social contract, while being a product of intentional deliberation, depends on conventions of fairness and just conduct. The connection becomes effective as soon as we introduce the possibility of social contract, not only to constrain behaviour, but also to modify interests. After reaching an agreement to reciprocally restrict behaviour to prevent conflicting

self-interest from arising, the members may be better able to observe the benefits of long-term consequences. Their consequential interests toward reciprocal behaviour may increase as they learn during the game. The game becomes developmental as experience together with expectations facilitates steps to a higher level of cooperation.

Conventions and interpretation

The development of conventions is linked with precedents and prominence (Schelling 1960, Lewis 1969). Interpreting the behavioural recommendations of conventions in specific situations may create problems even if the individual is procedurally motivated in finding the appropriate solution. The hierarchical structure of conventions does not necessarily help the task of interpretation. The individual may search for analogous conventions applied in situations somehow resembling the one at hand, or she may resort to a more general convention that applies through a number of dissimilar situations. For instance, a general convention of 'finders — keepers' that provides a moral argument for first possession is clear as a principle, but less so in empirical terms. Depending on a more precise convention of proper behaviour when finding money on the pavement, the finder may either consider herself the first possessor or not.

Consider open source property rights conventions against the finders — keepers convention. It seems morally plausible to argue that an individual obtains the property right to an unowned resource by mixing her mental and physical labour with it (Locke 1986, Hume 1969, Rothbard 1982, 33). According to this Lockean idea, nobody is in the position to simply pronounce legal ownership to a vast area of land without indicating a differential relation to it by, e.g., fencing and cultivating it. Analogously, one who is the first to pick up driftwood on an unowned shore has the right to claim the ownership title to the findings because no other principle offers more prominent justification (Sugden 1986, 95). The Western tradition of property rights is largely consistent with this principle.

The initiator of an open source software project is clearly the prominent candidate to claim ownership title to the project. The potential acquirer of an orphaned project needs to signal loudly her intentions, in order to make sure that the finders — keepers principle is applicable. In the same vein, forking is intuitively morally wrong because it violates the finders — keepers principle.

Open source development demonstrates something that seems to violate the finders-keepers principle, however. After a project has been developing for a period of time, it may turn out that someone outside the project management has put mental and physical labour into the project to a degree that might contest the right of the initial owner. The finders — keepers principle does not necessarily provide a clear-cut solution because, on the one hand, the initial owner has a strong entitlement, but on the other hand, new extensions and modifications can be viewed as new, hitherto unowned elements whose moral entitlement should go to the developer.

Examining open source conventions on ownership against the background of finders — keepers provokes a conjecture about an

inherent tendency of open source development to dissolve. The realisticness of this conjecture depends on the relative strengths of finders — keepers and open source property right conventions. The inherent morale in finders — keepers deals with balancing effort with entitlement. The more effort one puts into an unowned resource, the more justified a property right claim is. The open source convention of retaining the property right with the project initiator may contradict our interpretation of justice when contributions and efforts flow from the group at large. If this is so, our interpretation of finders-keepers is closer to what I suggested above, that the creation of a modification or extension is perceived *per se* as justified basis for ownership.

Later developments in open source software suggest a tendency toward disintegration and toward the proprietary model. Instead of putting efforts to the development of one Linux operating system, the community has offered dozens of commercial Linux versions. Their prices have risen to almost the same level as Microsoft Windows, their major closed-source rival.

Objective knowledge, modularity and project management

The objective knowledge aspect of open source software is clearly an unintended element. That source code functions as a coordinative language and as a functioning object at the same time, enhancing communication even though these functions have not been deliberately designed from the communication point of view.

Technological modularity demonstrates both potentially intentional and unintended elements. When Linus Torvalds in the early 1990s started developing the Linux kernel, he probably did not have technological modularity as one of his prime goals. Technological modularity may often be the result of purposeful deliberation, but it may also grow more organically during development. Irrespective of the degree of intent, technological modularity enhances communication as developers do not have to control the whole system at once. They can focus their communication to a limited set of features they want to develop. Another communication-aiding aspect of technological modularity is the coordinative function of shared interfaces. They delimit ways of communication and reduce the demand for versatile exchange of ideas. When all parties share an interpretation of the central aspects of an interface, they do not have to test the extent to which other parties share this knowledge (disregarding the fact that discrepancies in their interpretations may occur).

The unilateral right of the project initiator to function as the sole selector seems intriguing as it does not necessarily convey the conventionally desirable picture of functional efficiency. If the conjecture of this paper holds that selection is not the central issue in open source software development, since what matters most is the continuous inflow of variations and discoveries, then the connection between being the initiator of a project and receiving the property right to the whole project through convention appears potentially unjustified from the functional efficiency perspective.

Conclusions

This paper has suggested that in open source software development conventions play an important role in defining property rights. Social contracts perform an equally important function in preventing conflicting interests from destroying the cooperative mode of interaction. The open source software itself brings elements of objective knowledge and technological modularisation that enhance communication and coordination. All these elements together reduce the need for managerial control regarding both coordination of knowledge and provision of incentives.

In this model, intentional elements do not seem to receive any apparent priority. It is recognised, however, that the design of the initial social contract plays a central role in facilitating open development. Without its restrictions to non-reciprocal behaviour, open source software would hardly have developed to what it is today. On the other hand, it is equally important to recognise the source of social contract. The designers did not genuinely discover the purpose of the Copyleft, instead, they codified something that was already there in the form of earlier conventions of the software developers' community. By setting up the Copyleft terms they wanted to continue what they perceived as beneficial development which was under attack by the introduction of the proprietary model.

The dependency of social contract upon convention becomes apparent in the establishing process of a social contract. The Copyleft would have been impossible to establish as a social contract unless the members perceived its terms as fair and beneficial to development. Although social contract is conceptually a product of intentional design, its content is so strongly based on spontaneous development of conventions that it becomes difficult to distinguish what parts of its content are *not* already established by surrounding conventions.

A central question concerning the future of open source development is: which one becomes the prevailing social contract, Copyleft or OSD? If Copyleft wins out, then open source development has better chances to remain genuinely open, at the cost of foregone profits from the proprietary model. If the OSD/BSD becomes the social contract, it may enhance the destruction of open source development because the BSD license does not prohibit changes in the license itself, even though the consequence might be the transformation from open to closed source.

The option to take open source private and make profit has several consequences. (1) The existence of the option *per se* changes incentive structures as the members understand the dynamics of prisoners' dilemmas. 2) Opportunities for defection lead to changes in expectation about how other members will behave in the future. (3) The changed expectations reinforce incentives to defect. An important aspect in this process of incentive change is that the triggering element does not have to be a real event. The fact that an option exists may be enough to bring reluctance toward contributing to development that is vulnerable to defection. Another important aspect in this development has to do with

reference point consideration. If Copyleft did not exist as the initial social contract, the members would not perceive OSD/BSD as a potential deterioration of cooperation.

REFERENCES

BSD license, at http://www.opensource.org/licenses/bsd-license.html , retrieved 28 April 2001.

Crane, D. (1972) *Invisible Colleges, Diffusion of Knowledge in Scientific Communities*. Chicago: University of Chicago Press.

Cusumano, Michael A. (1997) How Microsoft Makes Large Teams Work Like Small Teams. *Sloan Management Review*, Fall, 9-20.

GNU General Public License (GPL), at http://www.opensource.org/licenses/gpl-license.html , retrieved 28 April 2001.

Huberman, B. A. and Tad Hogg (1995) Communities of Practice: Performance and Evolution. *Computational and Mathematical Organization Theory*, 1, 73-92.

Hume, David (1969 [1740]) *A Treatise of Human Nature*. London: Penguin Books.

Langlois, R. N. (2000) Modularity in Technology and Organization. Paper presented at 'Austrian Economics and the Theory of the Firm' conference, August 19-17, 1999, Copenhagen Business School, second draft.

Lave, J. and E. Wenger (1991) *Situated Learning: Legitimate Peripheral Participation*. Cambridge: Cambridge University Press.

Locke, John (1986 [1690]) *The Second Treatise on Civil Government* (first published in: Two Treatises of Government [1690]). Buffalo, NY: Prometheus Books.

Lewis, D. (1969) *Convention: A Philosophical Study*. Cambridge, MA: Harvard University Press.

OSD (Open Source Definition), at http://www.opensource.org/docs/definition.html , retrieved 28 April 2001.

Perens, Bruce (1999) The Open Source Definition, in Di Bona, Chris and Sam Ockman (eds) *Open Sources: Voices from the Open Source Revolution*. O'Reilly & Associates.

Popper, Karl R. (1972) *Objective Knowledge. An Evolutionary Approach*. Oxford: Clarendon Press.

Raymond, Eric S. (1998) 'Homesteading the Noosphere', at www.tuxedo.org/~esr/writings/homesteading/homesteading.txt , retrieved 10 March 2000.

Raymond, Eric S. (1999) 'The Magic Cauldron', at www.tuxedo.org/~esr/writings/magic-cauldron/magic-cauldron.txt retrieved 10 March 2000.

Rothbard, Murray N. (1982) *The Ethics of Liberty*. Humanities Press: Atlantic Highlands.

Sanchez, Ron and Joseph T. Mahoney (1996) Modularity, Flexibility, and Knowledge Management in Product and Organizational Design. *Strategic Management Journal*, 17, 63-76 (Winter Special Issue).

Schelling, Thomas (1960) *The Strategy of Conflict*. Cambridge, MA: Harvard University.

Stallman, Richard (1999) The GNU Operating System and the Free Software Movement, in Di Bona, Chris and Sam Ockman (eds) *Open Sources: Voices from the Open Source Revolution*. O'Reilly & Associates.

Sugden, Robert (1986) *The Economics of Rights, Co-operation and Welfare*. Oxford: Basil Blackwell.

Wow, 1 June 2000, at http://www.wow.fi/WOW/161700070054370014662973 93?path=talous/juttu&document_id=164412 , retrieved 1 June 2000.

Wow, 28 Nov 2000, at http://www.wow.fi/WOW/161700040054370014662973 93?path=talous/juttu&document_id=216395 , retrieved 28 April 2001.