Due July 30th, 2017

Legal Aspects

License Clearance in Software Product Governance

Dirk Riehle, dirk@riehle.org, http://osr.cs.fau.de

Nikolay Harutyunyan, nikolay.harutyunyan@fau.de

Abstract

Almost all software products today include open source components. However, the obligations that open source licenses put on their users can be difficult or undesirable to comply with [25] [14] [20]. As a consequence, software vendors and related companies need to govern the process by which open source components are included in their products [21] [7]. A key process of such open source governance is license clearance, that is, the process by which a company decides whether a particular component's license is acceptable for use in its products [19] [4] [15]. In this article, we discuss this process, review the challenges it poses to software vendors and provide unanswered research questions that result from it.[1]

1. License compliance

A legally valid software product complies with the licenses of all the open source components included in the product [19]. An open source license provides rights such as free (as in cost) use of the software in exchange for the fulfilment of obligations [14] [21]. Failure to meet these obligations leads to a legally invalid product. Some of these obligations could lead to intellectual property (IP) loss for the software vendor [18] [25] [14] [20].

1.1 License obligations

Consider the following three example obligations [7] [18]:

- *License file provision.* The most common obligation is to provide the license file of each open source component that comes with the product.

[1] This article is a follow-up to the NII Shonan Meeting on "Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability" where the first author was tasked with summarizing research questions in the domain of open source license clearance and software supply chain management.

- *Copyright notice provision.* Another common obligation is to provide all copyright notices from all files of each open source component.
- *Offer to provide source code (Copyleft[2]).* The typical Copyleft obligation is to either provide the product source code outright or to make a written offer to provide it upon request.

Some obligations are easy to comply with, and some are not. Some obligations are unproblematic, and some are highly undesirable from the intellectual property (IP) perspective of the vendor [25] [13] [17].

We therefore classify license obligations into three main types.

- *Unproblematic* (easy to comply with and unproblematic from an IP perspective). An example is the license file provision.
- *Difficult-to-comply-with* (difficult to comply with, but unproblematic from an IP perspective). An example is the complete copyright notice provision.
- *Undesirable* (from an IP perspective). For many, but not all, business models, an example is the obligation to provide source code outright or to offer to provide the source code.

1.1.1 Potentially difficult-to-comply-with license obligations

Whether an obligation is easy to comply with or not depends on various issues. For example, with improved tools, some obligations that are difficult to comply with today may become easy to comply with in the future.

Consider the case of the obligation to provide all copyright notices from all files of the original open source code. In theory, if all files were available and with adequate tool support, it would be possible to compile a document with all copyright notices.

However, this is based on the premise that the origin of every line of source code is known and has been documented. There is no guarantee for this. Developers easily and often copy code from the web and may have pasted code from one component into another without properly documenting it. Without such documentation, it is nearly impossible to determine the original source and therefore nearly impossible to comply with its license obligations.

1.1.2 Potentially undesirable license obligations

Whether a particular obligation is undesirable from the perspective of a vendor depends on the vendor's intellectual property strategy, which in turn depends on its business model. A traditional closed source vendor, for example, deriving significant revenue from license fees, may not want to be forced to license out their IP because of a Copyleft obligation [1].

Examples of license obligations often considered undesirable are:

- *Written offer to provide source code (Copyleft).* If this clause triggers, the vendor has to provide the source code outright or to provide the source code upon request under the Copyleft license [13] [3],

[2] Free Software Foundation, What Is Copyleft? at https://www.gnu.org/licenses/copyleft.en.html

thereby losing exclusive usage rights, among other downsides.

- *Patent retaliation clause.* This clause, if triggered, usually withdraws the right to use the open source component or the patent or both and thereby renders the product legally invalid, if the vendor enforces patent rights against someone else (the specifics depend on the license).
- *Lack of patent grant.* Some older licenses do not include a patent grant [28] [17]. Thus, any use of the open source component in a product exposes the vendor to a potential patent enforcement action by a patent holder who contributed an implementation of the patent to the open source component.

A firm that is earning its living by providing services and support for open source components may not worry about Copyleft but rather develop all software in the open. Depending on the warranties and indemnification the firm provides to its customers, however, it may worry about other issues like lack of patent grants.

1.2 License strategy

A rational software vendor can only accept components with unproblematic licenses into their products.

If the vendor were to accept a difficult-to-comply-with license, it might not be able to comply with the obligations and therefore end up with a legally invalid product. This opens the door for the original copyright holders to sue the vendor for license violation [1]. The Software Freedom Conservancy, a not-for-profit foundation, funds such lawsuits with the goal of enforcing license compliance. Also, developers exist, who pursue such a strategy for personal enrichment [19]; the details of the legal strategies are not of concern here.

If the vendor were to accept an undesirable obligation, the vendor might face a situation in which recipients of the software product insist on the vendor complying with the obligation. The vendor might decide to comply and face the consequences, for example, loss of exclusive rights to the intellectual property it created, or the vendor might decide to fight the request in court, leading to legal costs, lost management attention, and loss of reputation, among other downsides [19].

As a consequence, a software vendor needs to make sure that only open source components with unproblematic licenses are used in a product. This specific process of clearing a suggested open source component for use in a product is the license clearance process [19] [4] [15]. License clearance is part of open source governance, which is part of overall product governance.

2. Product governance

Product governance is the governance of all involved parties, their roles and responsibilities, as well as their processes and practices over the course of the product's life. It is mostly a product management task, but also involves engineering management and software architecture. Open source governance is that part of product governance that is concerned with the use of, contribution to, and leadership of open source software projects as they are relevant to the vendor's product.

Open source license clearance is one process of open source governance, and the main concern of this article.

However, to understand license clearance, we first need to understand the complexity of software products and how open source software makes it into a product.

2.1 Product architecture

Software products and most software components are built from other software components. As a consequence, a software product can be viewed as a graph of interconnected software components and fragments. The properties of the constituent parts of a product and their relationships are all relevant to product governance and need to be modeled precisely. Capturing this information is a precondition for achieving license compliance [8] [9], that is, correctly fulfilling all the obligations that the use of open source components puts onto the software vendor.

2.1.1 The component graph

Figure 1 illustrates the architecture of a product as a code component graph. The final product is shown at the top, and it depends on (incorporates and uses) various other components. These other components may have been developed by the vendor or they may have been sourced from a third party. Closed source vendors and open source projects are both viable sources of third-party components.

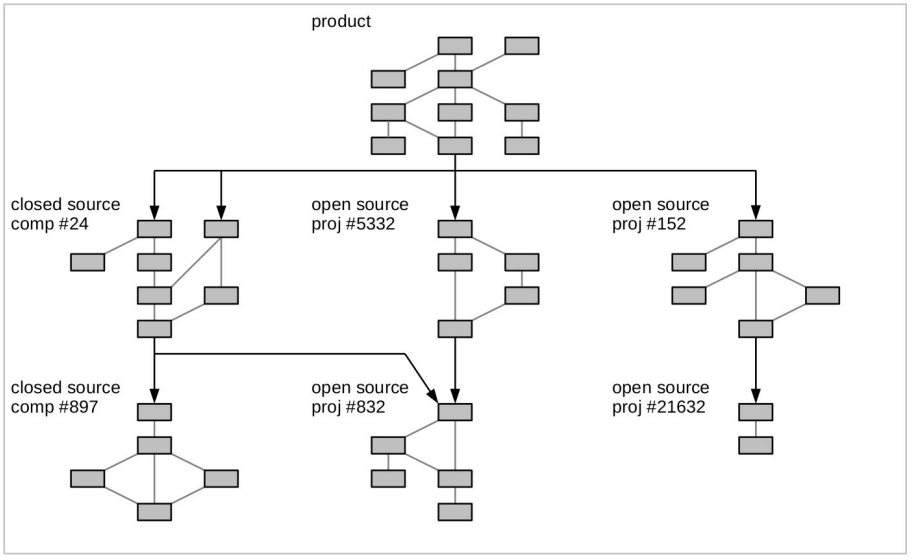

Figure 1. The architecture of an example product from the code component perspective

Example properties of interest for a given code component include

- *its license(s),*
- *any known vulnerabilities, or*
- *its export restrictions [22], for example, due to cryptography algorithms.*

An important view of the code architecture are management domains, which cluster components by their managers, that is, closed source vendors or open source project communities. A management domain corresponds with the traditional notion of a (possibly multi-component) third-party code component. Typically,

but not always, those who manage such a domain also own the copyright.

Figure 2 illustrates these management domains. Components of the same management domain usually, but not necessarily, have the same license. For example, the OpenJDK project, delivers many components, but most importantly the core runtime library needed by any Java application. This large library aggregates many other components of varying but compatible open source licenses.

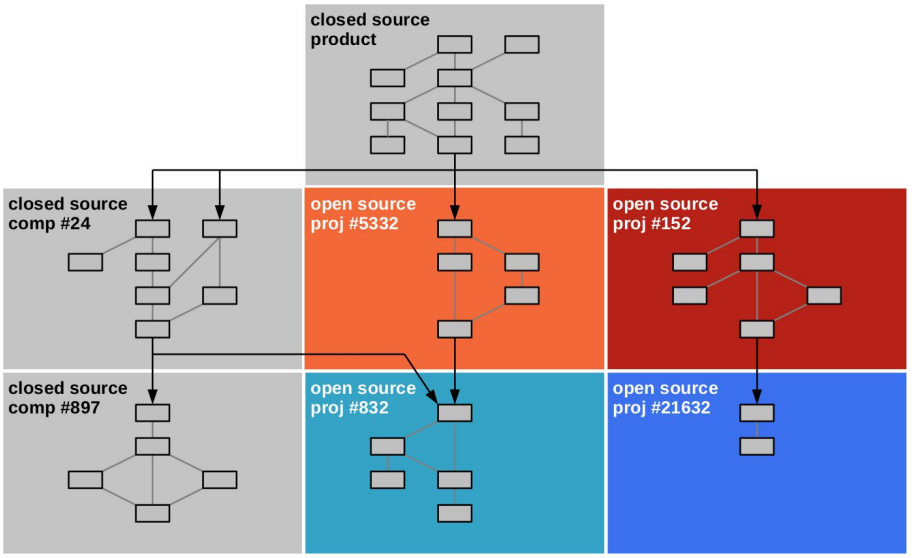

Figure 2. The code architecture of Figure 1 scoped by management domains

2.1.2 Component relationships

Viewing a product's code architecture as a graph of interdependent components requires engineering managers, software architects, and developers to be clear not only about which components to use (the nodes) but also to be clear as to how they relate (the edges). This is particularly important, if the component relationship crosses from one domain into another. From a license compliance perspective, understanding the component and fragment relationships is critical to making good decisions during the license clearance process. Depending on the type of relationship, license obligations may or may not apply [5].

Examples of relationship types are

- *the statically imported library,*
- *the dynamically loaded library, and*
- *the web services call.*

Also, copying code from the web or other places and pasting it into a software component introduces a dependency of the component onto some other party's intellectual property. Search engines, discussion forums, and question-answer websites for programmers make copying and pasting code easy today and it constitutes a frequent occurrence. Product governance policies may prevent this for closed source code, but open source projects typically do not have such provisions in place.

2.1.3 Code architecture model

Traditional modeling tools for software architecture do not support management domain views of code component architectures. Such a view, however, is often provided by tool vendors specializing in license compliance.

Still, most vendors, if they track the code component architecture for license compliance purposes at all, maintain a spreadsheet with the components, their licenses, and other meta-data. From this spreadsheet, the so-called bill of materials, license compliance artifacts like license texts and copyright notice compilations can be generated.

In theory, any component could provide its meta-data so that build tools could collect all relevant information and build the bill of materials automatically [9]. Sadly, this is not being done widely. As a consequence, most companies maintain a product's bill of materials by hand.

2.2 Make or buy decisions

From a software vendor's perspective, most components will be sourced from third parties, where third party providers can be other companies or open source projects. Free (as in cost) open source software is a great value proposition for software startups, but even established software vendors benefit from the cost reduction of using high quality components for free [2].

The main decision, whether to make or buy a particular software component for use as part of a product, is a product management decision. The driving criterion is whether the software component will in any way support the competitive differentiation of the product in the marketplace or not. If the component is not competitively differentiating, it should probably be sourced from a third party.

If no such component exists, the company may have to develop the component itself, but typically should do so as an open source component to harness the benefits that an engaged open source community can bring [24] [23] [10]. These benefits are

- *maturing the component faster,*
- *helping recruiting new and competent employees, and*
- *improving employee loyalty.*

2.3 The software supply chain

Software vendors need to look at their product's code component architecture and their sourcing of not-competitively differentiating components as a form of software supply chain management. They need to evaluate third parties as suppliers of components towards sustainability, quality, and costs, among other criteria.

Third party suppliers can be commercial companies or open source projects. Companies may be providing closed source components or they may be providing open source components with additional (to-pay-for)

services like warranties or support.

The supply chain view of a code component architecture naturally leads to supplier tiers, with the first tier of suppliers having a direct relationship with the vendor, and tiers further removed having an indirect relationship with the vendor. Still, the actions of tier 2 or higher suppliers directly impact the vendor's product [26]. Figure 3 illustrates the tier-view of the code component architecture.

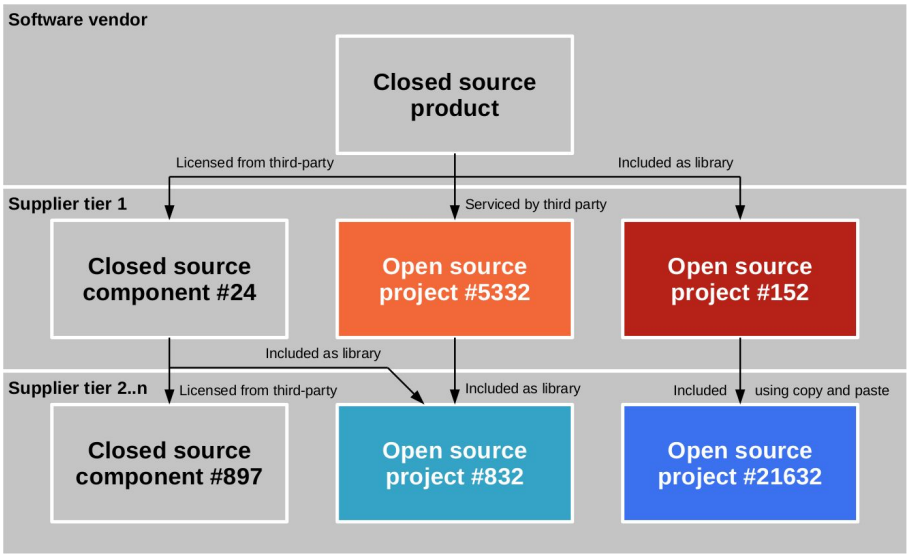

Figure 3: The supply chain perspective of the code component architecture of an example product

A direct relationship with a supplier allows a vendor to enforce their license strategy. For example, in a contract with a closed source supplier, the vendor may be able to specify that only unproblematic open source be used. Or, when choosing an open source component, the vendor may select only components with an unproblematic license.

However, contracts or declarations don't necessarily guarantee that reality conforms to them.

For example, a commercial supplier may promise, by contract, that no open source is used in their products. However, they may fail to enforce proper governance processes that ensure that developers do not copy open source code or include open source components in their components. Delivered as a binary, it may be difficult for the vendor to determine whether the supplier is meeting its contractual obligations.

Also, the declared license of an open source component may not necessarily be the real license of the component. Developers may have copied code from elsewhere that forces a license change, but may have failed to declare it. Or, some licenses conflict with each other, leading to software that cannot be used legally [5]. While the source code is available for analysis, determining any such license violation or confusion is not trivial.

Tiers 2 and higher may present the same problems to their next lower tier, compounding the effect on the vendor as the final user of the software.

As a consequence, many lawyers believe that little legally valid software is left on the market. They assume that so much copying and pasting has taken place that all software has been tainted. Without knowing what's in their software, a vendor cannot be license compliant and hence cannot ship a legally valid product.

The difficulty of determining unwanted code goes both ways: The vendor may find it difficult to determine, but so does the original copyright holder, who might have standing to sue. This mitigates the risks expressed through this otherwise bleak assessment.

2.4 Complete and correct bills of materials

To be able to make an informed decision and to ensure license compliance, a vendor therefore needs to receive or develop a complete and correct bill of materials for a supplied component. Both industry and open source communities have woken up to this challenge and are trying to address it.

The first step is to have a standard format for a bill of materials that expresses what is included in a component. For this, the Linux Foundation has sponsored the creation of the Software Package Data Exchange (SPDX) standard [27] and tools for processing the standard [19].

SPDX is rapidly evolving. SPDX compliant documents provide information about what is contained within a software package, including the license information of a contained component, who created the component, its version, etc.

A bill of materials also needs to be complete and correct. To this end, any open source project needs to exercise good open source governance. Guidelines of varying quality exist on the web [12] [6].

The Open Chain Project of the Linux Foundation is trying to address this problem by providing guidance to software vendors and open source projects on how to have good open source governance [16].

3. License clearance

License clearance is short for license clearance process. It is the process of reviewing and deciding upon requests to include third-party components, in particular open source components, in products. Typically, this process is part of the overall open source governance and compliance efforts of a company [7] [19] [9] [25] [14].

3.1 Process preconditions

The license clearance process has to have, at a minimum, the following three key components [7] [19].

- *A responsible person.* Someone needs to be tasked with the license clearance process. This person or post also needs to be known for being responsible for this process, and managers and developers need to have been educated to go to this person with any license clearance questions they may have.
- *A decision strategy.* The responsible person needs to know how to decide on a request to include an open source component in a vendor's product. For this, they need the license strategy and all necessary expertise. They may have to work with additional experts, for example, the vendor's legal counsel.
- *Escalation powers.* Finally, the responsible person needs the power to enforce its decisions, typically

by escalating a denied inclusion request that is getting ignored through the legal department to higher managerial levels in the company.

Vendors with state of the art processes typically will have established some sort of open source program office or open source competence center, whose responsibilities include open source governance and hence the license clearance process [7] [19] [11].

3.2 The clearance process

The clearance process itself can get complicated, but doesn't have to. We have identified the following common best practices (in no particular order):

- *Black lists and white lists.* With some licenses, the decision can be made quickly and independently of context. For example, the AGPLv3 license is typically not acceptable and should be black-listed, while the Apache License 2.0 is typically unproblematic and can be white-listed [9] [19].
- *Planned integration in products.* Sometimes, the context determines whether a particular component can be used. Depending on the embedding of the component in the product, unwanted obligations may not apply, in case of which the use of the component is unproblematic [9] [5].

  To make this decision, a model of the product architecture, as described in the previous section, is needed. A software architect needs to maintain the model to demonstrate to the license clearance process owner that the desired use of a component is unproblematic.
- *Review of license conflicts.* Some licenses conflict with each other and hence the components of these licenses cannot be used in the same product [5]. Using the model of the product architecture that we introduced, the process owner can check for such conflicts.
- *Component repository.* For efficiency reasons, the vendor may maintain an internal repository of component versions that have previously been accepted for inclusion in products. This is an advanced form of white-listing, making the use of open source components a self-service process.

  Since security vulnerabilities may not be known at the time of including a component in the repository, all white-listed components need to be monitored for newly discovered vulnerabilities and reevaluated in the light of any new information.

  A side-effect of providing a component repository is enhanced security. Developers should use components from the internal repository rather than a public one, reducing the attack surface for anyone trying to harm the vendor's products.
- *Component tracking.* Components in products need to be tracked. The first step is to maintain the product architecture model. The second step is to continuously review new information about the components embedded in the vendor's products.

New information may be problems with the license, new known vulnerabilities, or increased legal activities for the component. The vendor needs to react to such information, for example, by removing a component or upgrading the product to a new version of the component.

4. Research questions

The base of any license clearance is a complete and correct product architecture model. To build this model, the following challenges need to be mastered:

- How to receive a complete and correct bill of materials for an open source component?
    - How to represent this bill of material?
    - How to automatically generate the bill of material from project artifacts?
    - How to identify post-facto that code has been copied into a component from elsewhere?
- How to motivate an open source project community to clean up its code?
    - How to motivate an open source project community to create a bill of material?
    - How to motivate an open source project community to apply good open source governance?
- How to represent and work effectively with the product architecture model?
    - How to automatically generate complete and correct license compliance artifacts?

With a complete and correct product architecture model in place, the following challenges can be addressed:

- How to determine whether a particular license combination is legally valid?
    - How to completely and correctly model license obligations and their combination?

Finally, the vendor faces the challenge of ensuring the model conforms with the source code:

- How to ensure developers follow a proper license clearance process?
    - How to make the license clearance process known and understood?
    - How to ensure developers take the license clearance process serious?
    - How to make the license clearance process effective and not a burden?

5. Acknowledgments

We would like to thank our colleagues Daniel German and Matti Rossi for the discussions and collaboration at the 2017 workshop on FLOSS ecosystems at Shonan Village, Japan. We also would like to thank Maximilian Capraro, Shane Coughlan, Michael Dorner, Monika Schnizer, and Axel Teichert for their feedback on this article.

6. Bibliography

[1] Carver, B. W. (2005). Share and share alike: Understanding and enforcing open source and free software licenses. Berkeley Technology Law Journal, 443-481.

[2] Fitzgerald, B. (2006). The transformation of open source software. MIS Quarterly, 587-598.

[3] Free Software Foundation (2007). GNU General Public License: Version 3, 2007, at http://www.gnu.org/licenses/gpl.html

[4] German D., & Di Penta M. (2012). A Method for Open Source License Compliance of Java Applications. IEEE Software, 29 (3), 58-63.

[5] German, D. M., & Hassan, A. E. (2009). License integration patterns: Addressing license mismatches in component-based development. In Proceedings of the 31st International Conference on Software Engineering (pp. 188-198). IEEE Computer Society.

[6] GitHub (2017). Open Source Guides at https://opensource.guide/

[7] Haddad, I. (2016). Open Source Compliance in the Enterprise. The Linux Foundation.

[8] Helmreich M., & Riehle D. (2012). Geschäftsrisiken und Governance von Open-Source in Softwareprodukten. In Praxis der Wirtschaftsinformatik (HMD 283) 49. Jahrgang, 17-25.

[9] Hemel A., & Coughlan, S. (2017). Practical GPL compliance. Linux Foundation. 43-47.

[10] Henkel, J. (2004). Open source software from commercial firms–tools, complements, and collective invention. Zeitschrift für Betriebswirtschaft, 4, 1-23.

[11] Hewlett-Packard Development Company L.P. (2007). Best practices in open source governance. White paper.

[12] Jensen, C., & Scacchi, W. (2010). Governance in open source software development projects: A comparative multi-level analysis. Open Source Software: New Horizons, 130-142.

[13] Kennedy, D. M. (2001). A primer on open source licensing legal issues: copyright, copyleft and copyfuture. Louis U. Pub. L. Rev., 20, 345.

[14] Laurent, A. M. S. (2004). Understanding open source and free software licensing: guide to navigating licensing issues in existing & new software. O'Reilly Media, Inc.

[15] Link, C. (2010). Patterns for the commercial use of open source: legal and licensing aspects. In Proceedings of the 15th European Conference on Pattern Languages of Programs (p. 7). ACM.

[16] Linux Foundation (2017). The Open Chain Project at https://www.openchainproject.org/

[17] Mann, R. J. (2005). The Commercialization of Open Source Software: Do Property Rights Still Matter?. The University of Texas School of Law. Law and Economics Research Paper No. 58.

[18] McGowan, D. (2001). Legal implications of open-source software. U. Ill. L. Rev., 241.

[19] Meeker, H. J. (2017). Open (Source) for Business: A Practical Guide to Open Source Software Licensing (2nd ed.). CreateSpace Independent Publishing Platform.

[20] Meeker, H. J. (2008). The open source alternative: understanding risks and leveraging opportunities. John Wiley & Sons.

[21] Nadan, C. H. (2001). Open source licensing: Virus or virtue. Tex. Intell. Prop. LJ, 10, 349.

[22] Pearson, H. E. (2000). Open source licenses: Open source—the death of proprietary systems?. Computer Law & Security Review, 16(3), 151-156, Page 156.

[23] Riehle, D. (2009). The commercial open source business model. Value Creation in E-Business Management, 18-30.

[24] Riehle, D. (2007). The economic motivation of open source software: Stakeholder perspectives. Computer, 40(4).

[25] Ruffin, C., & Ebert, C. (2004). Using open source software in product development: A primer. IEEE software, 21(1), 82-86.

[26] Schöttle, H., & Steger, U. (2015). Managing Open Source Software in the Corporate Environment. *Computer Law Review International*, *16*(1), 1-7.

[27] Stewart, K., Odence, P., & Rockett, E. (2011). Software package data exchange (SPDX) specification. International Free and Open Source Software Law Review, 2(2), 191-196.

[28] Zhu, S. (2007). Patent rights under FOSS licensing schemes. Shidler JL Com. & Tech., 4, 4-13.