A Method for Open Source License Compliance of Java Applications

Daniel M. German, University of Victoria

Massimiliano Di Penta, University of Sannio

// *Kenen is a semiautomatic approach to open source license compliance for Java components that uses component identification, provenance discovery, license identification, and licensing requirements analysis.* //

OPEN SOURCE IS transforming the way software is built. The availability of artifacts (source files, components, libraries, and so on) from open source systems promotes reuse. Any software developer can quickly search the Internet and find a component to reuse, download it, and incorporate it into the software that he or she is creating. Because such reuse could violate the license under which the open source code has been distributed, the organization's legal department should carefully monitor it. In such a context, a major challenge is to know what open source is being used (if any) and how.

Open source license compliance (OSLC) is the process of ensuring that an organization satisfies the licensing requirements of the open source software it uses, whether for its internal use or as a product (or part of one) that it develops and redistributes.

Building on our previous work,[1–5] here we introduce readers to OSLC challenges and provide guidelines on how organizations can control and mitigate the legal risks associated with open source reuse.

OSLC Challenges

Ideally, an organization should prepare itself and its developers for the challenges of open source reuse.[6] Manuel Sojer and Joachim Henkel conducted a survey of several hundred developers and discovered that reusing open source is now common,[7] but software developers frequently don't understand the associated legal risks, and their organizations lack policies to guide them.[8] Sojer and Henkel also investigated whether different countries' legal systems (such as common law or civil law) or spoken languages (potentially affecting license understanding), affected code reuse; however, their results didn't provide any support to that conjecture.

Here, we'll define open source software as software that's licensed under an open source license. An open source license makes source code available for creating derivative works based on it. (According to the US Copyright Act, a derivative work is a work "based upon one or more pre-existing works.") The Open Source Initiative (OSI) has a set of characteristics that an open source license must fulfill to be OSI approved (see http://osi.org). To date, OSI has approved 69 different licenses, such as the GNU General Public License version 2 (GPLv2) and version 3 (GPLv3), the Apache Public License version 2.0 (APLv2), and the MIT license.

An open source license is a vehicle for the licensor to grant certain rights to the licensee that would otherwise be forbidden, such as the right to make copies and the right to create derivative works. In exchange for these rights, the licensee should abide by the requirements that such a license imposes.

Every open source license can be

0740-7459/12/$31.00 © 2012 IEEE

SOLVING THE LICENSE INCOMPATIBILITY PROBLEM

The proliferation of open source licenses, each with its own set of requirements and grants, frequently makes it impossible—in theory—to combine components with two or more different licenses. This problem is referred as the *license incompatibility problem*. Licensor and licensees have created ingenious solutions to this problem.[1]

For example, Mozilla code—originally licensed under MPLv1.1—couldn't be combined with code under the GPLv2 (the two licenses are incompatible). To solve this problem the Mozilla Foundation chose to relicense Mozilla under a "Disjunctive Licensing"—that is, under three licenses (the GPLv2, LGPLv2.1, and MPLv1.1)—and let the licensee choose one of them.

Another example is the "exception," where the licensor adds an addendum to a license (such as the GPL) that permits certain uses that would otherwise be forbidden. One instance of this is Oracle's FOSS License Exception to the GPLv2, stating that code licensed under any of 26 licenses (listed in the exception) can link to a GPLv2-licensed library to which it applies. This allows code under the PHP license, which isn't compatible with the GPLv2, to link to MySQL connect libraries. When Oracle (then Sun Microsystems) was considering licensing Java under the GPLv2, concerns arose that any Java program could be considered a derivative work of the Java SDK. To address this issue, the Free Software Foundation created the Classpath exception; when a Java library is licensed under the GPLv2 with the Classpath exception, the program that links to it is considered a derivative work of the library, but such a program can be distributed under any license (open source or not).

Reference

1. D.M. German and A.E. Hassan, "License Integration Patterns: Addressing License Mismatches in Component-Based Development," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), IEEE CS, 2009, pp. 188–198.

modeled as a set of grants. A grant is a right given by the copyright owner. Each grant requires the licensee to satisfy one or more requirements; the requirements can be modeled as a conjunction of predicates.[1] For example, APLv1.1 requires the licensee to include the component's copyright notice along with the binaries that incorporate the software. In this case, the grant is allowing the creation and distribution of binaries based on the software, and the requirement is the inclusion of the copyright notice. If the copyright notice is included with the binaries, the condition is satisfied, and the licensee receives the grant.

Open source licenses vary substantially in the constraints they impose. Any licensee must satisfy all the requirements for each license of the software it reuses. If the licensor can't satisfy them, then it can't reuse the open source in question. For example, a licensor that wants to distribute software under only a proprietary license will not be able to link to an open source library licensed under the GPLv3. The GPLv3 requires, as a condition for the grant of creating derivative works, that the derivative work can only be licensed under the GPLv3; it's not possible to satisfy this condition and, at the same time, license the software under a proprietary license (see the "Solving the License Incompatibility Problem" sidebar).

Reuse Methods

There are two main methods of open source reuse: copy-and-paste and component-based. Copy-and-paste methods involve copying sections of source code from an open source system. In component-based reuse, the open source system is used as a black box, becoming a module of the system being created. Component-based reuse can take different forms, such as libraries that are linked into binaries, executables required for the functioning of the system (such as language interpreters), and code generators (including compilers).

Legal Risks

Many open source licenses (such as the GPL family, which includes the GPLv2 and the GPLv3) distinguish the requirements they impose on a grant to create a derivative work and those they impose on other grants, such as making copies of the component and further distributing them. We leave to the courts the legal question of whether a software system is a derivative work of a component it reuses, and anybody who incorporates open source into commercial software should seek legal advice.

Even copying a few lines of code can be a legal risk. In one copy-and-paste case, a US Court of Appeals declared that just a few lines of copied code

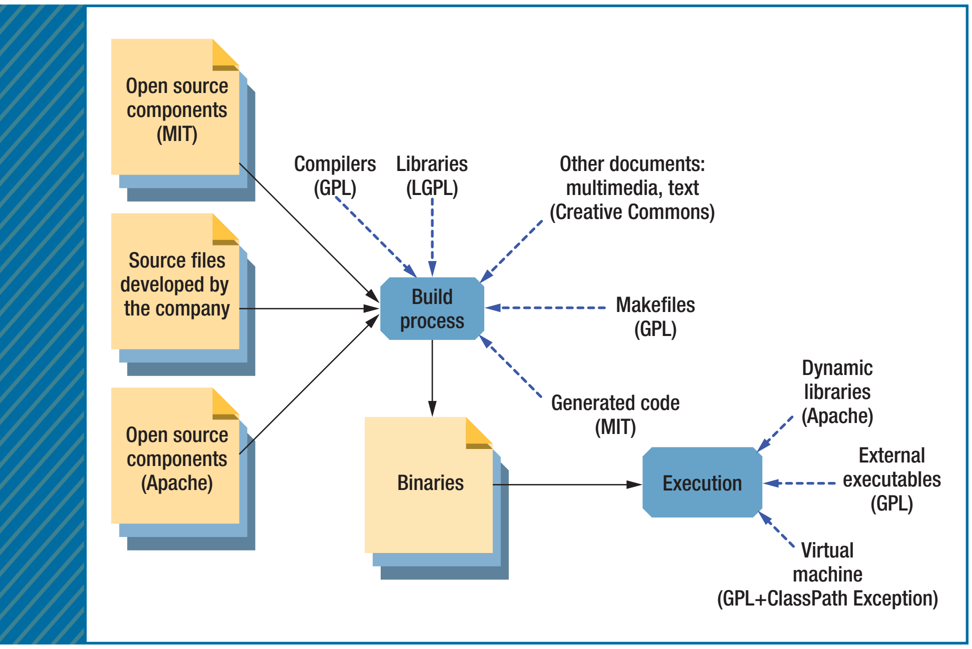

Open source components (MIT)
Source files developed by the company
Open source components (Apache)
Compilers (GPL)
Libraries (LGPL)
Other documents: multimedia, text (Creative Commons)
Build process
Makefiles (GPL)
Generated code (MIT)
Binaries
Execution
Dynamic libraries (Apache)
External executables (GPL)
Virtual machine (GPL+ClassPath Exception)

FIGURE 1. A proprietary software system might require many different open source systems with many different open source licenses.

could be a copyright infringement if the code is of "sufficient importance."[9]

The legal risks of component-based reuse are harder to evaluate. The method in which the component is integrated has a big impact in determining if a system is a derivative work of the component. It's always useful to contact the author and ask for his or her opinion on whether the intended use would create a derivative work (even if this opinion is not legally binding). For example, the Free Software Foundation (FSF, the creator of the GPL family of licenses) says that when a system links to a GPL-licensed library (either dynamically or statically), the result is a derivative work of the library. However, if the component executes in its own execution space (via a fork or sys.exec), then the system is not a derivative work. (This allows Apple's Mac OS, a product under a proprietary license, to include FSF open source tools such as gcc and emacs that are licensed under the GPLv3.) Linus Torvalds, the main author of the Linux kernel, has clarified that any program that uses the kernel via system calls is not its derivative work.

Reuse Policies

Today, all organizations should retain legal advice and create policies regarding the reuse of open source. Policymakers should address potential issues by

- appointing an open source officer who oversees the use of open source within the organization;
- stating under what circumstances open source reuse is allowed (copy-and-paste or component-based methods);
- training staff regarding open source licensing and reuse;
- creating a repository of preapproved components that can be reused;
- defining procedures for the approval of new open source components;
- defining procedures to document and verify how open source is being incorporated into a product;
- establishing an approval process that clears the release of a product that reuses open source; and
- defining procedures to guarantee that the organization satisfies the requirements imposed by the reused component in products (such as making its source code available for download or including copyright notices in its documentation) at both the time of its release and after (and for as long as it is necessary to fulfill these requirements).

Figure 1 illustrates how different open source and commercial products, with many different licenses, can be involved in creating a proprietary system. Open source can be part of its source code, or it can be involved in its building and running.

Kenen: An OSLC Process for Java

Today, any organization producing software should verify that it satisfies the license constraints of the open source it uses. Kenen is a semiautomatic process to help organizations in their OSLC for Java development (see Figure 2). It comprises four main stages.

Creation of a Repository of Preapproved Components

To ease the development of systems that fulfill licensing compliance, it's a good policy to have a repository of preapproved reusable components (including components that are developed in-house, that are licensed by the organization from third parties via contracts, and that are open source). An organization needs to bootstrap such a repository by scanning its current systems to identify any open source it's using.

Provenance Discovery

What open source is the organization using? This question can be explored at both the source code and the binary levels. In previous work,[5] we created a provenance analysis tool called Joa,

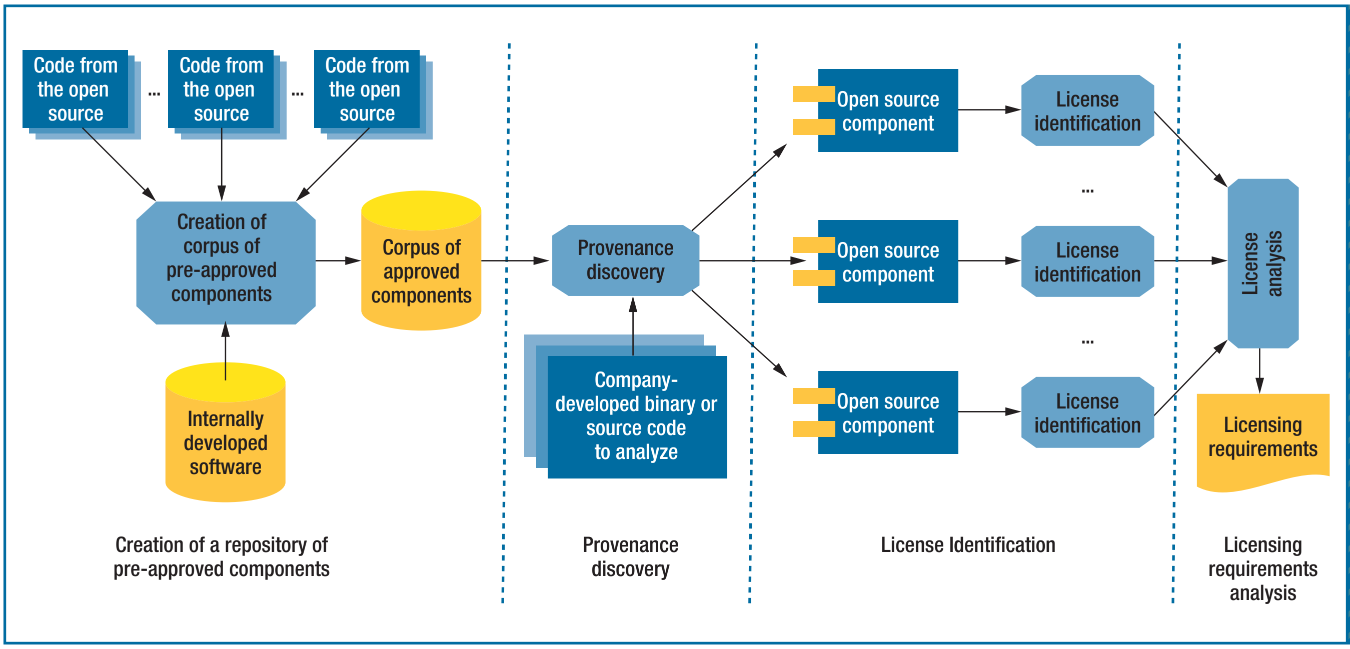

Code from the open source
...
Code from the open source
...
Code from the open source
Creation of corpus of pre-approved components
Corpus of approved components
Internally developed software
Provenance discovery
Company-developed binary or source code to analyze
Open source component
Open source component
Open source component
License identification
License identification
License identification
...
...
License analysis
Licensing requirements
Creation of a repository of pre-approved components
Provenance discovery
License Identification
Licensing requirements analysis

FIGURE 2. An overview of Kenen. Before the analysis is done, Kenen creates a repository of preapproved components, identifies the license of each of them, and analyzes these licenses to determine if the requirements are legally compliant.

which can find matches of both source and byte code Java files in a large corpus of candidates. Joa compares a class's type signature with those of potential candidates and suggests potential matches. It can find matches as long as the type signatures and their methods haven't diverged too much; otherwise, clone-detection tools will work instead. If Joa finds no match of a given source (.java) or binary (.class) file against this repository, then we assume that the product contains non-approved source code. In this case, it's useful to run Joa against a universal repository. Unfortunately, maintaining a universal repository is very difficult. For our experiments, we used Maven2 (http://mirrors.ibiblio.org/pub/mirrors/maven2) as a corpus of open source. Maven2 is a repository of source code and binaries that has been created to support Maven, a dependency manager for Java applications.

Joa finds the best match for a Java Archive file (JAR) based on two metrics: the similarity index and the inclusion index. The similarity index is the number of classes that share the same signature between the JAR file (the subject) and the candidate JAR (among those in the corpus), divided by the size of the union of class signatures of the subject and the candidate. The inclusion index of a subject and a candidate is the number of classes in the subject that are also in the candidate, divided by the size of the candidate. The similarity index can help us identify almost identical copies; the inclusion index can help us identify when a given JAR is a proper subset of another.

License Identification

For any open source component that we reuse, we should identify its license. We should do this for the specific version matched (the one copied to be reused), because its license might change over time (for example, the current version of the component might have a different license from the one copied). Identifying the license of an open source component isn't always trivial.[4] One of the main challenges is that open source components don't have a uniform way to document their licenses. Some efforts, such as the Software Package Data Exchange (see the "Software Package Data Exchange" sidebar) are attempting to standardize this information and to share it between organizations. To identify the license of the files that compose a given Java component, we use Ninka, a pattern-matching-based tool that we developed in previous research.[3]

Licensing Requirements Analysis

The licensing requirements analysis is a manual process that should be performed by somebody with both software and legal expertise. How complex and time consuming this analysis is will depend on the number of open source components involved and their licensing requirements.

The licensing requirements analysis answers two questions: first, is the overall license of the system compatible with the licenses of each of the

SOFTWARE PACKAGE DATA EXCHANGE

The Software Package Data Exchange (SPDX) is a standard format for the description of the license of a software system. Its goal is to document, in a well-defined format, the license of a system and its files and their licenses. Many organizations and individuals have participated in its definition, including HP, Canonical, Motorola, Black Duck Software, Texas Instruments, and Daniel German, one of the authors of this article. SPDX has also started to standardize the names of the most common open source licenses. Version 1.0 of SPDX was released in August 2011. The long-term goal of SPDX is to become a standard for the exchange of bills of materials that accompany any given system, indicating all its components and every one of their licenses. This will simplify licensing compliance analysis across the supply chain. For more information, visit www.spdx.org.

components it uses? This requires evaluation of whether the overall system's license contradicts any of the license requirements of each component it uses. If there is even one contradiction, the system can't be licensed to others.

The second question is whether all the requirements stated in those licenses are fulfilled. The automated analysis of any possible license, with the aim of identifying its requirements and how they are triggered, is a nontrivial problem. Licenses are written in legal terms and are intended to be interpreted by lawyers, not computers. To further complicate the problem, requirements might be triggered (or not) based on the jurisdiction of the licensor, the licensee, or both. Ideally, a lawyer who specializes in this area of law should interpret every license, and he or she would determine its requirements and the conditions that it imposes on the organization wanting to reuse it.

A Case Study

To illustrate the effectiveness of Kenen, we analyzed the provenance and licensing constraints of a software system (an editor of music files, which we will not name for confidentiality reasons), for which we only had access to its binaries.

Implementing Kenen

We wanted to know if Kenen could verify whether an application used open source, and if so, whether it satisfied its licensing constraints. As mentioned earlier, an organization should have a corpus of preapproved components. In our study, we used Maven2 instead. As of July 2011, it included 523,930 archives (.tar, .java, .zip, and so on) totaling 275 Gbytes. Using Joa, we extracted the signatures of every .class and .java file (27,851,789 files). A one-time preprocessing analysis of the repository created a database of signatures. This step took approximately 325 hours on a typical desktop computer.

The application was composed of 57 different JAR files; Joa identified the source of 27 as present in Maven2. (Joa took few seconds to run for each JAR in a typical laptop computer with a solid state drive.) Of the remaining 30, not a single class was present in Maven2. We presume that the organization authored some of these JARs and that some open source isn't available in Maven2. It's hard to build a universal repository that contains every version of every open source repository artifact ever released.

For those JARs identified as being in Maven2, 16 had a similarity index of 1 (perfect matches). For the remaining nine, the median similarity was 0.72.

With only one exception, the base name of each subject JAR file (without version) matched the name of the best match (saxpath.jar matched saxpath-1.0-FCS.jar in Maven2); 60 out of 95 classes matched to izpack-uninstaller-1.0.0.0.jar (its best match). These facts give us confidence that our analysis was accurate. Table 1 shows some examples of the JARs in the application and their best matches.

Once we had the JARs' corresponding source code, we identified their licenses (some of which are summarized in Table 1). Identifying the licenses of each product depends on whether Ninka can identify the license. Ninka automatically identified 20 JARs' licenses. For the other seven, we had to manually analyze the files' license statements and the component's documentation. (This process took approximately four hours. In general, the duration of this process will vary according to factors such as the component's size and how well its license is documented.)

Using this information, we performed a license analysis. We determined that none of the component's licenses were incompatible with a proprietary license, but some of them required listing the copyright owners in "the documentation and/or other materials provided with the distribution." The analyzed system's license satisfied this condition by including within its distributed files a directory called "licenses," which listed the license (and therefore the copyright owners) of some components. The list of copyright owners of one of the components, however, was not present, even though it was required (xpp3_minb-1.1.4c.jar). We contacted the organization that authors the application, which quickly acknowledged the problem and will

TABLE 1

Provenance and licensing analysis of some of the JARs found in the proprietary application.

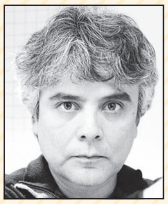
JAR	Size (no. of classes)	Best match	Size	Similarity	License
commons-io-1.4.jar	72	commons-io-1.4.jar	72	1.000	APLv2
saxpath.jar	15	saxpath-1.0-FCS.jar	15	1.000	*APLv1.1-type
jaxen-1.1.1.jar	197	jaxen-1.1.1.jar	197	1.000	BSD-3
swingx-ws.jar	134	swingx-ws-1.0.jar	134	1.000	LGPLv2.1
xpp3_min-1.1.4c.jar	3	xpp3_minb-1.1.4c.jar	3	1.000	APLv1.1-type
jide-oss.jar	414	jide-oss-2.4.8.jar	409	0.927	Free commercial use
swingx.jar	426	swingx-0.9.2.jar	422	0.785	LGPLv2.1

**APLv1.1-type refers to licenses that are similar, but not identical to the Apache Public License v1.1.*

address it in the next version of the product. This analysis took approximately three hours.

Today, developers can easily find and reuse open source components, even without management knowledge or authorization. To mitigate any potential legal risk, organizations should directly address OSLC issues by creating policies that clearly indicate if open source is allowed and the procedures to follow for its approval. A process such as Kenen is an important tool in verifying that these policies are being observed.

ABOUT THE AUTHORS



DANIEL M. GERMAN is associate professor of computer science at the University of Victoria, Canada. His research interests include open source legal compliance, open source software engineering, and software evolution. German has a PhD in computer science from the University of Waterloo, Canada. He is a member of the Legal Network of the Free Software Foundation Europe. Contact him at dmg@uvic.ca.

MASSIMILIANO DI PENTA is associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, reverse engineering, empirical software engineering, search-based software engineering, and service-centric software engineering. Di Penta has a PhD in computer engineering from the University of Sannio, Italy. Contact him at dipenta@unisannio.it.

References

1. D.M. German and A.E. Hassan, “License Integration Patterns: Addressing License Mismatches in Component-Based Development,” *Proc. 31st Int’l Conf. Software Eng.* (ICSE 09), IEEE CS, 2009, pp. 188–198.
2. M. Di Penta et al., “An Exploratory Study of the Evolution of Software Licensing,” *Proc. 32rd Int’l Conf. Software Eng.* (ICSE 10), IEEE CS, 2010, pp. 145–154.
3. D.M. German, Y. Manabe, and K. Inoue, “A Sentence-Matching Method for Automatic License Identification of Source Code Files,” *Proc. IEEE/ACM Int’l Conf. Automated Software Eng.* (ASE 10), ACM, 2010, pp. 437–446.
4. D.M. German, M. Di Penta, and J. Davis, “Understanding and Auditing the Licensing of Open Source Software Distributions,” *Proc. 18th Int’l Conf. Program Comprehension* (ICPC 10), IEEE CS, 2010, pp. 84–93.
5. J. Davies et al., “Software Bertillonage: Finding the Provenance of an Entity,” *Proc. Working Conf. Mining Software Repositories* (MSR 11), ACM, 2011, pp. 183–192.
6. C. Ruffin and C. Ebert, “Using Open Source Software in Product Development: A Primer,” *IEEE Software*, vol. 21, no. 1, 2004, pp. 82–86.
7. M. Sojer and J. Henkel, “Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments,” *J. Association for Information Systems*, vol. 11, no. 12, 2010, article 2.
8. M. Sojer and J. Henkel, “License Risks from Ad Hoc Reuse of Code from the Internet,” *Comm. ACM*, Dec. 2011, pp. 74–81.
9. N.J. Mertzel, “Copying 0.03 Percent of Software Code Base Not ‘De Minimis,’” *J. Intellectual Property Law & Practice*, vol. 9, no. 3, 2008, pp. 547–548.