Automated software license analysis

Timo Tuunanen · Jussi Koskinen · Tommi Kärkkäinen

Received: 18 August 2008 / Accepted: 25 May 2009 / Published online: 6 June 2009

Abstract Software license is a legal instrument governing the usage or redistribution of copyright-protected software. License analysis is an elaborate undertaking, especially in case of large software consisting of numerous modules under different licenses. This paper describes an automated approach for supporting software license analysis. The approach is implemented in a reverse engineering tool called ASLA. We provide a detailed description of the architecture and features of the tool. The tool is evaluated on the basis of an analysis of 12 OSS (open source software) packages. The results show that licenses for (on average) 89% of the source code files can be identified by using ASLA and that the efficiency of the automated analysis is (on average) 111 files per second. In a further comparison with two other open source license analyzers—OSLC and FOSSology—ASLA shows a competitive performance. The results validate the general feasibility of the ASLA approach in the context of analyzing non-trivial OSS packages.

用于识别代码的许可情况，而不是判断许可冲突

Keywords Software licenses · Software reuse · Open source software · Reverse engineering · Program comprehension

T. Tuunanen (✉) · T. Kärkkäinen
Department of Mathematical Information Technology, University of Jyväskylä, P.O. Box 35 (Agora), 40014 Jyväskylä, Finland
e-mail: timtuun@jyu.fi

T. Kärkkäinen
e-mail: tka@mit.jyu.fi

J. Koskinen
Department of Computer Science and Information Systems, University of Jyväskylä, P.O. Box 35 (Agora), 40014 Jyväskylä, Finland
e-mail: koskinen@jyu.fi

1 Introduction

The costs of software maintenance activities have traditionally been estimated at 50-75% of total software life-cycle costs (Lientz et al. 1978). Moreover, according to some studies (Seacord et al. 2003) the percentage related to maintenance and evolution is increasing, so the importance of this subarea is clearly great. According to Lehman's first law (Lehman et al. 1998), the software must be continually adapted or it will become progressively less satisfactory in real-world environments.

This is due to continuous changes in user requirements and in aspects of the technical environment. Successful systems inevitably change, due to pressures from changing requirements (Lehman et al. 1998). Software changes need to be planned and implemented in a controlled fashion, in order to avoid deterioration of the software quality due to undesired side-effects resulting from the changes. The typical problems which make changes harder include poor modularity, poor structure, and poor documentation. For example, instances of textually highly delocalized but logically strongly dependent parts of the program can lead to comprehension problems (Letovsky and Soloway 1986). Moreover, comprehension problems in turn make it harder both to fully and partially reuse software in general. Many legacy systems are also very large investments, containing invaluable business logic and knowledge. Because of this fact, there is a need to reuse their components.

Component-based software reuse is one way to reduce the problems of legacy system maintenance. Generally speaking, the reuse of well-defined and tested components can be particularly beneficial. Reverse engineering techniques can be used for retrieving information that is relevant to the maintenance, reuse, and comprehension of large-scale programs. Most reverse engineering tools are based on parsing the source code and providing abstracted views of system components and their interrelations. The provided views support the tool user in making the right choices and decisions concerning potentially reusable components.

逆向的方法类似分析 watermark

Open source software (OSS) development has some typical characteristics, such as licensing, which need to be considered when one is designing reverse engineering support for reuse. In fact, software license analysis is a pre-condition for legally taking components for reuse, or for modification for a specific purpose. The potential effects of having only limited rights to reuse and modify software components need to be taken into account in some manner, and for large systems this is a non-trivial process. This being so, it is helpful to have automated tools that can provide information regarding the use of the licenses and other general views of the software.

Detailed empirical results characterizing the reuse-based software development of large-scale systems have been provided by Selby (2005). That study showed the importance of some module design factors (including few calls to other system modules from a module and many calls from it to utility functions). The module implementation factors that characterize module reuse without revision were small size in source lines and (after normalization by size in source lines): low development effort and many assignment statements. The study also underlined the extent to which a low proportion of faults will minimize the need for revision of modules that are candidates for reuse.

In making decisions on the potential reuse of software components (and modules), it is necessary to take licensing into account (an aspect not investigated by Selby

2005). Ideally, a reuse-support tool or environment would provide a versatile profile of the candidate modules, regarding their reusability.

Software licenses are typically determined at module level. Module dependencies may introduce undesired coupling. As a general rule, excessively high coupling is often caused by the use of global variables (Yu et al. 2004) and by excessive procedure calls beyond module boundaries. File inclusions also create dependencies between source code files. These dependencies are typically scrutinized during the software build process. An additional problem for reuse and comprehension may be the breaking of modularization rules that are often otherwise relatively well followed at the software-build level (de Jonge 2005).

This paper describes an approach and a tool for the reverse engineering of software licenses, with a view to determining the intellectual property rights (IPRs) related to available (open source) software systems. The paper extends the previous work (Tuunanen et al. 2006) by describing the tool (including some new features) in a more detailed manner, and by providing a more detailed evaluation of the efficiency of the tool. The paper also includes a comparison between the tool and two other license analysis tools, regarding the features that they provide and the efficiency of the tools.

The paper is organized as follows. Section 2 first characterizes OSS development, and goes on to address software reuse and reverse engineering. Section 3 describes the characteristics of software licensing. Section 4 lists the user needs pertaining to license analysis, based on the previous sections. Section 5 describes an automated reverse engineering approach and its implementation, called ASLA (Automated Software License Analyzer), whose function is to retrieve relevant license information from source code modules. The approach and its implementation are not restricted to OSS. However, since OSS provides many licenses and versions for potential reuse, it is a natural environment for developing and testing the approach. Section 6 presents an evaluation of the efficiency and coverage of the license identification and analysis of the ASLA tool. Section 7 discusses the main related works and compares ASLA against two other tools—OSLC and FOSSology—in terms of satisfaction of user needs. Finally, Sect. 8 draws conclusions.

2 Context of the study

The maintenance and reuse of large-scale legacy software is demanding, and the problems are even harder when one is dealing with programs which are hard to understand. Large-scale and poorly structured programs present challenging cases. Maintenance is especially difficult if documentation is inadequate or misleading. In solving maintenance problems, the maintainers constantly need relevant pieces of information, related to information needs (Koskinen et al. 2004) that will be characterized in more detail in Sect. 2.3.

The information typically needed includes identifiers, definitions, calling dependencies, parts of the documentation, and so on. This information is retrieved from various information sources, including source code and documentation. In the case of OSS reuse, license information retrieval together with the proper management of intellectual property rights (IPRs) forms an essential part of successful project management.

When license analyzers are examined, it can be seen that they can be used as a support for the OSS reuse process. From the technical and licensing points of view, the procedure can be presented as follows:

1. Search and download the OSS package of interest.
2. Analyze the package by using a license analyzer.
3. Examine the package from the perspective of its functionality and its licenses.
4. Select the reusable components and attach the retrieved license information to them.
5. Integrate the appropriate components within the developed or maintained application.

In this setting, all license analysis tools attempt to reduce the time required to reliably examine potentially reusable components, as compared to performing the analysis without automated support. Since without automation this task can often be completed only by employing lawyers at huge cost, the potential financial benefits are clear. It is possible to set out some example cases that can follow from the license analysis: (1) an OSS package is licensed under a permissive license and can be used as part of a commercial software product; (2) an OSS package is licensed using a restrictive license (e.g. GPL), allowing it to be modified and used as a company's internal tool, but not allowing it to be redistributed outside the company; (3) some parts of an OSS package can be reused as part of commercial software, whereas other parts can not be reused. These outcomes are usually related to company policies; for example, some device manufacturers reuse GPL code and provide the source code with the device, whereas other companies prevent GPL usage altogether.

One of the main reuse-related problems is the identification and comprehension of relevant pieces of programs, and their dependencies in a specific maintenance situation. This problem is especially severe when one is dealing with so-called delocalized program plans (Letovsky and Soloway 1986), whose main characteristic is that the logically related program parts are for some reason textually dispersed. In this situation, one needs tool support to provide relevant information.

In this paper we apply reverse engineering to provide automated license identification and analysis. This is especially useful in supporting the reuse of OSS. In view of this, knowledge of various aspects related to OSS is required. In the following subsections we shall first discuss OSS (Sect. 2.1), software reuse (Sect. 2.2), and reverse engineering (Sect. 2.3).

2.1 Open source software

In the past few years, a wider research community has become increasingly aware of the contribution made by OSS development—to the software industry, to business and to society in general. Software engineering researchers explore OSS specifically with respect to development tools and methodologies, while organizational scientists and economists are keen to understand how the open source movement can lead large communities of people to help each other efficiently (Damiani et al. 2006). The term OSS involves two different purposes: (1) it is a new methodology for creating software, including new process models, validation methods and social aspects, and (2) OSS offers new data sources for software reuse and case studies. From our perspective as tool developers, OSS serves mainly as a free data source. However, the different aspects of OSS are also briefly described below.

OSS development process outputs have been studied, for example on the basis of a sample of 406 projects (Capiluppi et al. 2003). The most used languages according to this study were C, C++, Perl, and Java. The numbers of developers and subscribers for individual projects are typically low (Capiluppi et al. 2003). Hence, despite the large number of OSS projects, development efforts have focused on a few large projects such as Linux, Mozilla, and Apache (Mockus et al. 2002).

The definitions for OSS-related terminology are provided by Perens (2005). The OSS community provides a rich base of potentially reusable software. Unlike the more traditional closed source software, OSS can be freely used, modified, and redistributed. Source code is also freely accessible. Software vitality is based on active maintenance of OSS by the OSS community or by various companies involved. However, there exist over 50 different versions of OSS licenses as listed by Open Source Initiative (Opensource 2007).

There are differences between open source and closed source software development models. Some open source advocates claim that OSS development fosters faster system growth. This claim has not been supported by research (Paulson et al. 2004). However, the OSS development model does seem to support creativity, since even long-lasting OSS is constantly updated and enhanced (Paulson et al. 2004). It seems to be the case that if OSS projects succeed, it is not because they are simpler than closed source projects; in fact OSS projects seem to produce more complex code than closed source projects. On the other hand, even though OSS code is complex, it has fewer defects than the code in closed source projects because defects are found and fixed rapidly (Paulson et al. 2004). One point to be made is that OSS seems to be less modular than closed source software (Paulson et al. 2004). This lack of modularity—which is most probably due to the lack of a clear architecture planning and assessment phase—makes OSS less attractive from a reuse perspective.

One important aspect in OSS development is the need for greater maintainability. Based on an analysis of almost 6 million LOCs it was concluded that OSS development produces legacy systems in much the same way as closed source software development (Samoladas et al. 2004). It has been observed that 20% of the components will produce about 80% of the maintainability problems (Yu et al. 2004). This means that the risk-prone modules need to be identified and their modification and comprehension supported.

2.2 Software reuse

Software reuse has been practiced since the very beginning of programming. It enables developers to use past achievements, and it improves software productivity and quality (Selby 2005). More than a decade ago it was predicted that future breakthroughs in software productivity would depend on the software community's ability to combine existing pieces of software to produce new applications (Garlan et al.

1995). At the same time it was understood that efficient software reuse is difficult to achieve, especially in unanticipated scenarios, i.e. in situations where a programmer would like to reuse a piece of software that was not designed for reuse.

Software reuse relates to numerous artifacts in software projects. For example architecture, source code, design, documentation, and requirements can be reused (Frakes and Terry 1996). Since reuse promises improvements in software quality and productivity, there has also been active research in this field. During the past 20 years the research has focused, for example, on generators, design patterns, component factories, and domain engineering (Frakes and Kang 2005).

The emergence of OSS has had a significant influence on reuse possibilities. Reuse is built into the open source/free software ideology, as stated in Free Software Definition (FSD 2003): *Free software is a matter of the users' freedom to run, copy, distribute, study, change, and improve the software* (Perens 2005). Even though OSS packages are not necessarily designed with reuse in mind, OSS components are viable candidates for reuse, due to their abundance and easy availability.

An empirical study of key success factors in software reuse in general (based on 24 projects) was conducted by Morisio (2002). The success factors mainly included organizational and process issues. According to some studies (Morad and Kuflik 2005), reusing OSS can bring up new challenges: Does OSS support the specific product requirements? Was it developed and tested using the same development environment as is used by the potential reuser? What kind of support may be available for that software? How are new version releases handled? For the most part, these potential problems are also present when reusing proprietary software.

The role of licenses is also prominent. License information concerning the dependency of different modules provides the key meta-information when one is considering reuse in a proprietary environment. Component-based reuse of OSS is natural, because, for example, license information is typically bound to modules. Since code changes are possible, a more specific case is white-box reuse. It is generally acknowledged that understanding undocumented and complex source code is relatively difficult, but it is clear that good tool support reduces the comprehension problems. Reuse can be supported by identifying reusable component candidates, simplifying the license identification, and providing abstracted views of the relevant components and their interrelations.

black-box reuse需要逆向分析项目

2.3 Reverse engineering

Reverse engineering is the process of analyzing an existing system in order to identify the system's components and their interrelationships (Paakki et al. 1997), and to create representations of the system, possibly at a higher level of abstraction. It does not involve changing the subject system: it is a process of examination (Cross et al. 1992). Reverse engineering is the main automated approach for satisfying the information needs of maintainers (Koskinen et al. 2004). Part of the information needed by maintainers can be retrieved via automated analysis. In an ideal situation all the required central information would be easily retrievable through the use of a reverse engineering tool.

Cross et al. (1992) describes two subareas of reverse engineering that are particularly useful in facilitating program understanding: redocumentation and design recovery. Redocumentation is the process of creating representations of a subject system that relate to some characteristics of the system. The resulting forms of representation are considered alternative views, intended to enhance the understandability of the system. These views include data flow, data structure, and control flow diagrams. Redocumentation can be characterized as the post-creation of the artifacts which should have been created during the earlier development process. Design recovery, for its part, is a more ambitious recovery of the systems function, purpose, or essence. Design recovery attempts to identify meaningful higher-level abstractions.

Many reverse engineering tools, such as those listed in Koskinen et al. (2004), have been designed to alleviate the typical comprehension problems of maintaining large software systems. Most of the tools focus on meeting the needs of programs written in C (and partly C++) (Bellay and Gall 1998). Some of the tools provide very versatile features in terms of their specific focus areas. For example, CodeSurfer (Anderson et al. 2003) provides sophisticated whole-program static analysis and program slicing. The results of the automated analyses of that tool are visualized and provided to the tool user, for example in the form of call graphs and dependency graphs. The tool can be used to support e.g. systematic software inspections. Reverse engineering tools can support many central software engineering tasks such as program comprehension, debugging, and impact analysis. For example, impact analysis (Lindvall 2003) is important in revealing the effects and scope of proposed changes and existing dependencies. Our own approach differs from that of typical reverse engineering tools in terms of its focus, which is the support of tasks related specifically to dealing with software licenses. These problems resemble more traditional comprehension problems, in the sense that both types of tasks require the identification and determination of the relevant components, and of the dependencies that exist between those components. Typically, there is a need for browsings or walk-throughs of parts of code which contain pieces of license information, and these also resemble reviews aimed at determining the quality and suitability of the code for the intended purpose.

Reverse engineering tools have been compared in many studies. The paper by Koskinen and Lehmonen (2008) lists six important comparative studies, including Bellay and Gall (1998). Reverse engineering tools typically extract the relevant information from the source code and store it in a program database. In most cases the extraction is achieved by calling up a parser component which is implemented according to the well-established conventions of compiler construction (Aho et al. 1986). Five-level classification of the information retrieval features of reverse engineering tools is provided by Koskinen et al. (2004). In the present paper, this classification will be used to organize the discussion of information needs and reverse engineering features (see Table 2). The levels of the model are: 抽象语法树，AST，源码的抽象表示

L1 Formation of basic level internal data structures (such as abstract syntax trees).
L2 Formation of higher abstraction level access structures (such as call graphs for tool users).
L3 Visualization of those structures.
L4 Information request and retrieval mechanisms.
L5 Navigation mechanisms within the formed access structures.

Koskinen et al. (2004) also compare the typical features of the main reverse engineering tools. These include variable references, cross-references, call graphs, module dependency graphs, data flow graphs (plus dependency graphs of interrelated pieces of documentation) and source code. For example, SNiFF+, which is compared to other tools in Bellay and Gall (1998), is a highly versatile tool that meets most of these needs. There are also some relevant studies based on structural program analysis and on text and documentation analysis, as listed in Koskinen (2000); these include automated software inspection (Anderson et al. 2003), and approaches operating at the intersection of data mining and reverse engineering (Andritsos and Miller 2001).

3 Software licensing

In the case of open source, individual programmers are responsible for creating software that does not violate IPRs. In the case of commercial projects, the project manager or separate IPR boards are usually responsible for making sure that project outputs are in order from the IPR point of view. License information retrieval is a relevant activity in projects which reuse OSS, whether they are commercial or open source.

确保项目产品符合IPR

Software reuse is very simple from the legal point of view, if a company or an individual reuses software for which it has copyrights. However, things change dramatically if one wants to reuse software made by others, since software is protected by copyright and possibly by patents. Without explicit permission, no person other than the copyright holder is allowed to copy, distribute, or make derivative works from the original work.

可以借用此描述

Software license refers to the kind of permission granted by a copyright or patent holder to use his or her intellectual property in a particular way (Rosen 2005). To illustrate the differences between different types of licenses, Table 1 presents the rights of the licensee in different cases. A typical proprietary end-user license agreement (EULA) will give the user merely a restricted right to use the software. For example, Microsoft Word and other typical end-user products have this kind of license. If proprietary software is to be reused as part of another program, the license must grant rights at least to copy and redistribute the work. This would be the case, for example, with Microsoft (2007).

Open source licenses are, however, different from typical proprietary licenses. Every open source license grants the following rights to the licensees (Rosen 2005):

1. Licensees are free to use the OSS for any purpose whatsoever.
2. Licensees are free to make copies of the OSS and to distribute them without payment of royalties to a licensor.
3. Licensees are free to make derivative works of the OSS and to distribute them without payment of royalties to a licensor.

Table 1 License categories

License	Usage	Redistribution	Derivative works
PROPRIETARY	Restricted	Not allowed	Not allowed
SHAREWARE	Restricted	Allowed	Not allowed
FREEWARE	Allowed	Allowed	Not allowed
OPEN SOURCE	Allowed	Allowed	Allowed

4. Licensees are free to access and use the source code of the OSS.
5. Licensees are free to combine the OSS and other software.

Based on the sample of 406 active OSS projects (Capiluppi et al. 2003), it appears that GPL is the most common OSS license (77%), followed by LGPL (6%) and BSD (5%). Other well known licenses include, for example, the Apache License (2007) and the Mozilla Public License (MPL 2007).

Nevertheless, the rights granted by all OSS licenses are merely principles that can be explicitly or implicitly expressed in the actual license text. Usually, OSS licenses include some additional restrictions or rights. On the basis of these restrictions (or lack of them), open source licenses can be categorized as being academic or reciprocal licenses (Rosen 2005).

- *Academic licenses* were originally created by academic institutions to distribute their software to the public. They allow the software to be used for any purpose whatsoever, with no obligation on the part of the licensee to distribute the source code of the derivative works. Anyone can take such software for any purpose, including the creation of proprietary works, without giving anything back to the open source community (Rosen 2005). Typical academic licenses are BSD (2007), MIT (2007), and Apache License (2007).
- *Reciprocal licenses* also allow software to be used for any purpose whatsoever, but they require the distributors of derivative works to distribute those works under the same license, with the condition that also the source code of those derivative works must be published. The GPL license is the archetypal reciprocal license. Reciprocal licenses, like academic licenses, contribute software to a public commons of free software, but they require also that derivative works should be placed in the same commons (Rosen 2005).

Licensing a derivative work that contains software under different licenses is a task frequently faced by most open source reusers. However, not all open source licenses are compatible with each other in the creation of derivative works. For example, GPL and MPL are incompatible with each other (Rosen 2005): you cannot make a derivative work that contains both GPL and MPL licensed code. In such a case, the programmer would violate one or other license. Hence, possible incompatibility problems must be located and solved before OSS can be reused efficiently. To make things even more complicated, even the binding mechanism between program components may determine whether licenses are compatible with each other. For example, according to many interpretations, LGPL licensed libraries can be linked to software, and that software can be licensed freely if linking is done dynamically (e.g. using DLLs in Windows or shared objects in *nix). But if LGPLd code is linked statically (for example if the library is included as part of the compiled binary file), the program must be licensed under LGPL. This difference between dynamic and static linking and its implications for license compatibility must be taken into account when one is conducting license analysis.

The license of an open source program can be indicated in different ways within the actual software package. The preferred way, and the one most often used, is to have a note in each source file indicating the license of the file, with a reference to the full license text. In the case of academic licenses, the full license text is often found in each source file, since the license text itself is relatively short. Usually the full license text can be found in a file called COPYING. However, there are often source files that have no indication of the license used. The typical interpretation in this situation is to apply the license found in COPYING, in the case that the file is in the same directory (or in some parent directory in case of a clear separate component).

4 User needs

As described above, OSS licensing is a somewhat complicated field, and one that requires attention in order to manage IPR risks in OSS related projects. Our goal is to support IPR-aware reuse of OSS packages through automated license analysis. Since provision of legal advice is restricted to lawyers in many countries, our goal as system developers is rather to provide an automated tool that will allow users to define the rules and the outcome of the actual analysis. The user needs described below are based on the practical problems of open source license analysis: identification of licenses and their dependencies and the need for further analysis of all other relevant issues that can not be handled automatically. We divide user needs into three categories: basic needs, interaction-related needs, and non-functional needs. At each subsection we shall list the main needs and give explanations of why they are important for viable license analysis support.

4.1 Basic user needs

The user needs for the basic functionality of the tool are as follows:

N1 Identification of the program modules (source files, build process outputs, and libraries) that are included in the program within a particular environment.
N2 License identification of each source file.
N3 Dependency identification of program modules.
N4 The formation of license compatibility rules.
N5 The identification of licensing problems (such as missing licensing information and licenses that are incompatible between modules).
N6 The visualization of license analysis results.

OSS reuse can be classified as belonging to two different approaches. Hence there can be either (1) use of the entire software package (i.e. full reuse) or (2) use of part of the software package as part of another program (i.e. partial reuse). In either case, the reuser of OSS must know the license(s) of the software that he or she is about to reuse, and the possible incompatibilities between the different licenses in the packages used (see Sect. 3). It is also relevant to the OSS reuser to know which parts of the program are actually used (N1) from the software package in a specific environment. This is firstly because reuse can be partial, and secondly because in the case of many software packages, large (usually environment dependent) portions of software are not used at all.

To make a reliable license analysis, the license analyzer needs to know the licenses of each individual source code file (N2), since a single incompatibly licensed file can prevent the reuse of the whole software package. In addition, the dependencies between these source files and the compiled objects (program modules) (N3) must be identified in order to conduct the license compatibility analysis. When build process outputs are identified, the information can also be used for component identification. This identification can give some clues about reusable components within a larger software package, and this becomes useful when one is considering partial reuse.

The identification of licenses and dependencies between files forms a basis for more advanced license analysis. As described in Sect. 3, different OSS licenses can be incompatible with each other. To detect the incompatibilities, the tool must have rules that define the compatibilities between the licenses (N4). Since our goal is to make the license analysis as automated as possible, the automated detection of potential licensing problems (N5) is one of our main concerns. This need is dependent on the first four needs (i.e. N1–N4), and it cannot be satisfied unless the information satisfying the first four needs is available. Naturally, when the license analysis is conducted, the results must be presented to the user (N6). Since the results include information about possibly thousands of source code files, the initial visualization cannot include all the relevant information. To get more detailed information about the results of the analysis, user interaction is required (see below). Initial visualization can show, for example, that the analysis has been concluded, and that the analyzed component has (or does not have) potential licensing problems.

4.2 Interaction-related user needs

When the license analyst is conducting the analysis for a software package, he or she often encounters situations where fully automated license analysis is impossible. Hence, one must be able to explore the analysis that has been conducted more thoroughly, in order to find the reason for this. This, plus the necessary further interaction between the tool and the tool user, create the following interaction needs:

N7 Browsing of the results of the license analysis.
N8 Manual determination of the source code license.
N9 The addition of license identification templates.
N10 The visualization of license compatibility rules.
N11 The definition of license compatibility rules.
N12 Statistical information on the analyzed software package and the license analysis.

Since source code packages can contain thousands of source files, the results of the license analysis cannot be presented to the user statically. Because of this, the user must have the possibility to browse the individual files and their dependencies (N7). In addition, the user needs to be able to see the license of each individual file and the source code of the file. It is also necessary that the user should be able to get the list of licenses found in the package.

The license analyst often encounters a situation in which the license of the source file is indicated in a way that is not known in advance. In most cases programmers who write OSS use predefined templates (Opensource 2007) to indicate the use of a specific license. Unfortunately, this is not the case in all software packages. This

Table 2 Relations between information retrieval support levels and functional license analysis user needs

Support level	User needs
L1 Formation of basic level internal data structures	N1 Identification of the program modules (source files, build process outputs, and libraries) that are included in the program within a particular environment
	N4 The formation of license compatibility rules
	N9 The addition of license identification templates
L2 Formation of higher abstraction level access structures	N2 License identification of each source file
	N3 Dependency identification of program modules
	N5 The identification of licensing problems (such as missing licensing information and incompatible licenses between modules)
	N11 The definition of license compatibility rules
	N12 Statistical information on the analyzed software package and the license analysis
L3 Visualization of these structures	N6 The visualization of license analysis results
	N10 The visualization of license compatibility rules
L4 Information request and retrieval mechanisms	N8 Manual determination of the source code license
L5 Navigation mechanisms within the formed access structures	N7 Browsing of the results of the license analysis

means that it has to be possible to add new identification criteria for licenses (N9). In some cases one also needs to be able to determine the licenses manually (N8).

As described in Sect. 3, the compatibility of the OSS licenses can be interpreted in various ways. The actual interpretations of how different dependencies affect the license analysis results must be presented to the user (N10). Since we are not lawyers, it is left to the user to define the compatibility rules between the different licenses (N11). It is also necessary that the statistical information that can be determined from the analysis process should be summarized for the user (N12).

Section 2.3 introduced a five-level classification of the information retrieval support levels of reverse engineering tools. The user needs, as described above, are linked to these support levels in Table 2.

4.3 Non-functional user needs

There are also some non-functional (quality) needs for the license analysis. These relate to:

N13 The efficiency of the license analysis process, in comparison to manual analysis and existing tools.
N14 The coverage of the license identification.

The central aim of automated license analysis is to improve the efficiency of the analysis as compared to manual analysis. It is therefore also important that the performance of the automated analysis is at an appropriate level (N13). The system must scale up for software packages of a practical size, and adding new license templates should not slow down the overall process significantly. Nevertheless, scalability is not the main priority of the system.

The identification coverage of the license analysis must be as wide as possible (N14). However, this aspect can vary significantly, since many packages contain files that have no indication whatsoever of the applicable license. Our goal is to identify the license of each file that can possibly be identified, using both automated and interactive techniques.

5 ASLA

The most primitive method of license analysis is to read through all the source code files and to check the licenses of all files manually. This technique can be made more effective by automated license identification of source code files (using text searching techniques) and by providing information about file dependencies. In this section we present our reverse engineering approach, which uses the ASLA (Automated Software License Analyzer) tool for this purpose.

5.1 System architecture

ASLA system is implemented for the Linux operating system using the Java programming language (version 1.6.0_03). It consists of 36 classes and its size is about 5000 LOC. ASLA supports all programming languages that can be compiled, and for which dependency information files can be written, through the use of GCC (GNU Compiler Collection) (GCC 2007). The system is tested using GCC version 4.1.3. It also uses ar (an archive tool) and ld (a linker), based on GNU Binutils (2007) version 2.18.50.

Figure 1 shows the system architecture as a UML class diagram. The system utilizes build process outputs produced by GCC, GNU ar, and GNU ld. These external programs store information on dependencies between program parts. ASLA reads these dependency information files (DIFs), and performs the actual license analysis for the source files listed in them. This process is described in Sects. 5.2.2 and 5.2.3.

Figure 1 describes the most important classes in ASLA, and the external programs and files that are used by the system. The classes of ASLA are divided into two packages: *Back-end classes* and *User interface classes*. The back-end classes correspond to levels L1 and L2 of the information retrieval support levels of reverse engineering tools (see Sect. 2.3) dealing with the creation of data structures. The user interface classes correspond to levels L3–L5, covering the interaction with tool users.

5.2 Features

Figure 2 shows the relationships between functional user needs and the features of ASLA. The features are described in the following subsections in the order of the figure. They are designed to meet the user needs introduced in Sect. 4.

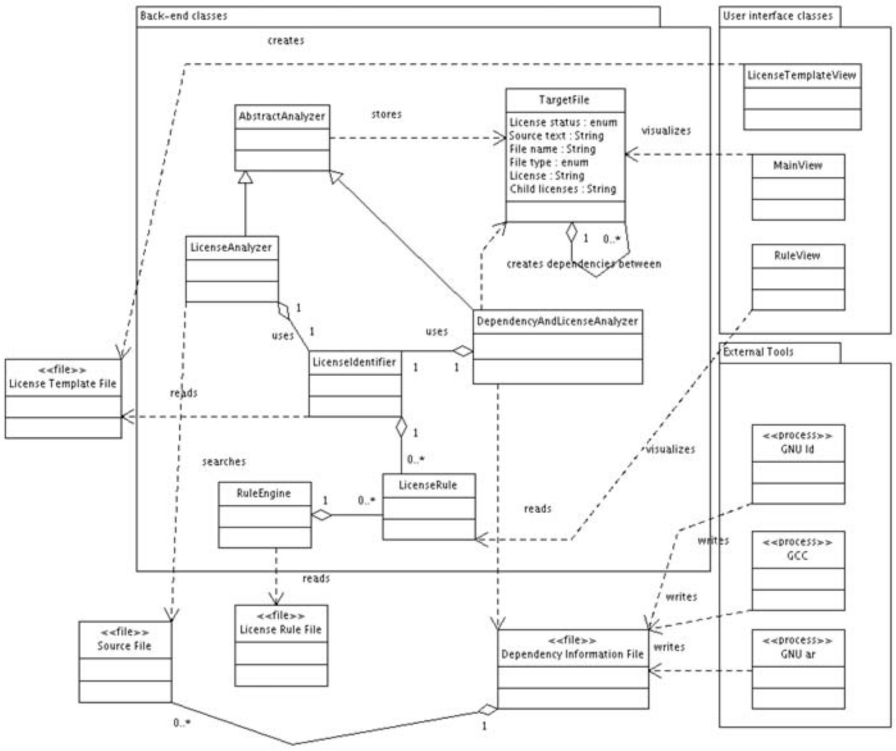


Fig. 1 ASLA system architecture

5.2.1 System-level features

The most typical usage of our tool is for dependency and license analysis (F1.1) (see Fig. 2). This user-initiated functionality fulfills all the basic user needs and combines the results of other features, including: basic data structure creation, automated license identification, and license compatibility checking. These functional features are described in more detail in the following subsections.

ASLA also makes it possible to identify licenses without dependency analysis (F1.2). This license identification of source code files is designed to be used with software packages that cannot be compiled using GCC. We can also identify licenses of source files that are programmed using e.g. Perl, Java, or even HTML. When the license identification of the source code files is conducted, ASLA searches for all the source code files and attempts to recognize their licenses.

5.2.2 Dependency analysis

The first task of the license analysis process from the system point of view is to identify all objects (source files, compiled objects, libraries, etc.) and dependencies between these objects. Dependency analysis forms a basis for license analysis via which

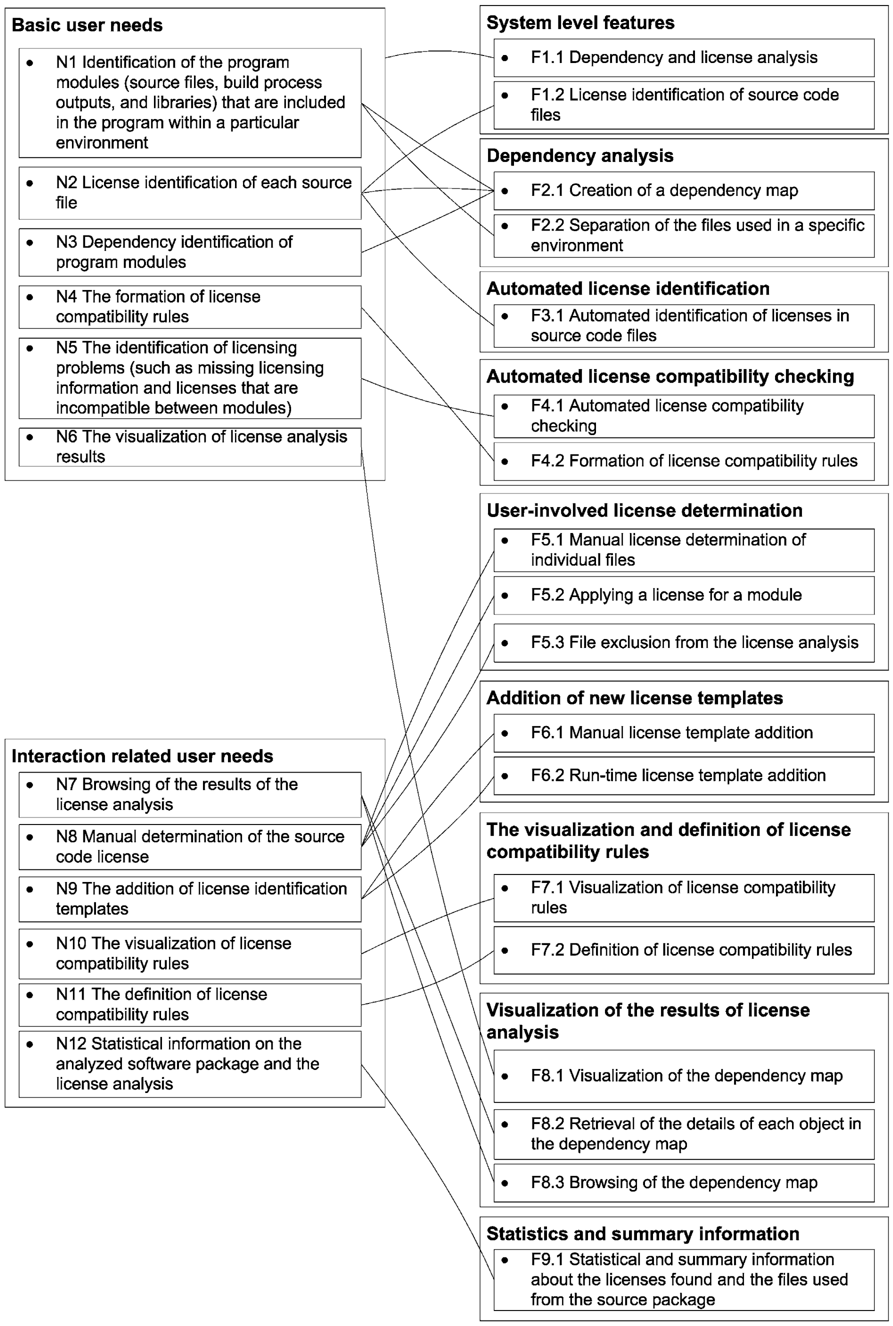

Basic user needs
N1 Identification of the program modules (source files, build process outputs, and libraries) that are included in the program within a particular environment
N2 License identification of each source file
N3 Dependency identification of program modules
N4 The formation of license compatibility rules
N5 The identification of licensing problems (such as missing licensing information and licenses that are incompatible between modules)
N6 The visualization of license analysis results
Interaction related user needs
N7 Browsing of the results of the license analysis
N8 Manual determination of the source code license
N9 The addition of license identification templates
N10 The visualization of license compatibility rules
N11 The definition of license compatibility rules
N12 Statistical information on the analyzed software package and the license analysis
System level features
F1.1 Dependency and license analysis
F1.2 License identification of source code files
Dependency analysis
F2.1 Creation of a dependency map
F2.2 Separation of the files used in a specific environment
Automated license identification
F3.1 Automated identification of licenses in source code files
Automated license compatibility checking
F4.1 Automated license compatibility checking
F4.2 Formation of license compatibility rules
User-involved license determination
F5.1 Manual license determination of individual files
F5.2 Applying a license for a module
F5.3 File exclusion from the license analysis
Addition of new license templates
F6.1 Manual license template addition
F6.2 Run-time license template addition
The visualization and definition of license compatibility rules
F7.1 Visualization of license compatibility rules
F7.2 Definition of license compatibility rules
Visualization of the results of license analysis
F8.1 Visualization of the dependency map
F8.2 Retrieval of the details of each object in the dependency map
F8.3 Browsing of the dependency map
Statistics and summary information
F9.1 Statistical and summary information about the licenses found and the files used from the source package

Fig. 2 Functional user needs of license analysis, ASLA's features, and their main relations (the *lines* indicate the features satisfying specific user need)

we make sure that the analyzed package does not include licensing problems. Dependency analysis also identifies potentially reusable components from a larger package (partial reuse). The dependency map is a data structure in which each of these files, either used or created during the build process, is stored (F2.1). Each object in the map has references to the objects that it is dependent on, and references to the objects that are dependent on it. The formation of the dependency map is achieved using dependency information files written by GCC, ar, and ld. Since GCC only produces information on source code dependencies (i.e. what source files include other source files), we use modified versions of linker (ld) and archive builder (ar) to collect information about the actual binary level dependencies (i.e. which object implements the function called from the source file). Dependency information files constitute base information for dependency map creation. Each dependency information file holds information about dependencies between objects, and in most cases there are several of these files. The dependency map is created by combining the information from these files.

ASLA is the only known license analysis tool that provides full information on build process outputs and their dependencies. In theory, build process outputs (and files included in those outputs) can be identified by reading build instruction files (e.g. Makefiles). However, that approach is very complicated, due to the fact that build instruction files can themselves be extremely complex. Our approach, in which dependency information files are used, is simple, and it performs well even with large software packages. One drawback is the need to compile analyzed packages; this affects the overall performance of the analysis as compared to the optimal case in which compilation would not need to be carried out. However, during the compilation process we can easily collect information on the dependency type (static or dynamic), and this is very useful in the actual license analysis, as described below. It should be noted that with the use of this technique it is not possible to identify run-time dependencies. Some software packages have, for example, plug-in architecture in which new features can be loaded during the run-time. ASLA is not able to identify the dependencies between these plug-ins and the software itself. Details of the dependency map creation have been provided in an earlier paper (Tuunanen et al. 2006) by the present authors. Each compiled object gets its license as a collection of its child licenses. During the dependency map creation, a special treatment of .lo files (created by GNU Libtool 2008), symbolic links, and duplicate files is required, to ensure the correctness of the dependency map.

Since we are using dependency information files, we are also able to get the list of files that are used in a particular environment, and to distinguish them from the files that are present but not used in the software package (F2.2).

5.2.3 Automated license identification

The goal of the license identification is to automatically identify the licenses of all source files. Automated license identification (F3.1) is achieved by using license templates given as regular expressions. Simple open source licenses, such as BSD and MIT, are often included at the beginning of each source code file (Opensource 2007). Another way to indicate the license that the source code is under is to make a reference to the license from the source code. This technique is usually used for open source licenses such as GPL, LGPL, and MPL (Opensource 2007) that have longer license text.

GPL (version 2) template that should be added to the beginning of each source file reads (partially) as follows:

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version [...]

license text matching

Identifying the licenses of source files that include pre-defined template or full license text should be fairly simple: one merely needs to match the pre-defined license text with the source code file. Unfortunately this kind of exact matching does not work very well with real source code files, for the following reasons:

1. The comment characters and the various kinds of white space characters prevent exact matching.
2. Many programmers modify the predefined texts, for example, by replacing *This program* (see GPL above) with the name of their software.
3. There are different published versions of the licenses. For example, LGPL was previously called GNU Library general public license and is nowadays GNU Lesser general public license. This change is also visible in the license notification attached to each source code file.

This means that there can be many slightly different texts within source code indicating the same license. Hence their recognition requires more sophisticated text matching techniques. In particular, regular expressions can be used to allow for white space characters, alternative words, and undefined characters. In our approach, each well-known open source license has its own regular expression, which is used for license matching. The license search template (regular expression) for LGPL version 2 and 2.1 is as follows:

is free software; you can redistribute it and/or modify (it)? under the terms of the GNU (Library)—(Lesser) General Public License as published by the Free Software Foundation; either version 2., or \\(at your option\\) any later version*
. is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE See the GNU (Library)—(Lesser) General Public License for more details*
You should have received a copy of the GNU (Library)—(Lesser) General Public License along with this (program)—(library); if not, write to the Free Software Foundation . .*USA*

The actual identification in ASLA is performed using Java *java.util.regex.Matcher* and
java.util.regex.Pattern classes as follows:

Pattern pattern = Pattern.compile(regexp, Pattern.MULTILINE);

Matcher matcher = pattern.matcher(sourceCodeString);
matcher.find();

Unfortunately reliable identification of the license of every single source code file is not possible; thus it must also be possible for the tool user to identify licenses manually. The features that require user involvement in license identification will be presented in Sect. 5.2.5.

5.2.4 Automated license compatibility checking

Automated license compatibility checking (F4.1) significantly reduces the work of manual license analysis, since the user is not required to manually solve the licenses and dependencies of each individual source file. To achieve extendable automated compatibility checking, our tool includes compatibility rules (F4.2) for each license. A license compatibility rule defines how two licenses co-operate with each other.

There are four possible states that can be defined in ASLA:

1. NOK (Not OK). Used in situations of clear incompatibility (e.g. commercial licenses and GPL).
2. Warning. Used in situations where the reuser is willing to receive notification on license usages that might be problematic but are not necessarily incompatible. (e.g. use of an original BSD license with an advertising clause).
3. OK. Used when licenses are compatible with each other.
4. N/A (not available). Means that no rule operating between the two licenses has been defined.

License rules are defined for both dynamic and static linking, since licenses can behave differently depending on the linking style (see Sect. 3). Rules also have a direction. The rule between two licenses is always defined twice, depending on which license is the parent license. For example, a rule between BSD and GPL will be different, according to how the parent file is licensed. In the case where the parent file is under GPL, a rule for BSD is OK, since files under academic licenses can be included in work under GPL license. However, if the parent file is licensed under BSD, a Warning or NOK should be used, since the viral effect of GPL comes into effect, and the parent file can no longer be licensed under BSD.

Compatibility rules are user definable, as will be described in Sect. 5.2.7. On the basis of these compatibility rules, ASLA will automatically check for incompatibilities between licenses in the course of license analysis, using the known information concerning the licenses of each individual file, and the dependencies between them.

To illustrate the functionality of this technique, let us consider the following example (Fig. 3). The software artifacts (files) are shown as rectangles and a dashed arrow points from file A to B, such that A is dependent on B.

We have three source files: *main.c*, *linked.h*, and *linked.c* that are compiled and linked. In this case we have the following dependencies:

- *main.c* is dependent on *linked.h*
- *linked.c* is dependent on *linked.h*
- *linked.o* is dependent on *linked.c*

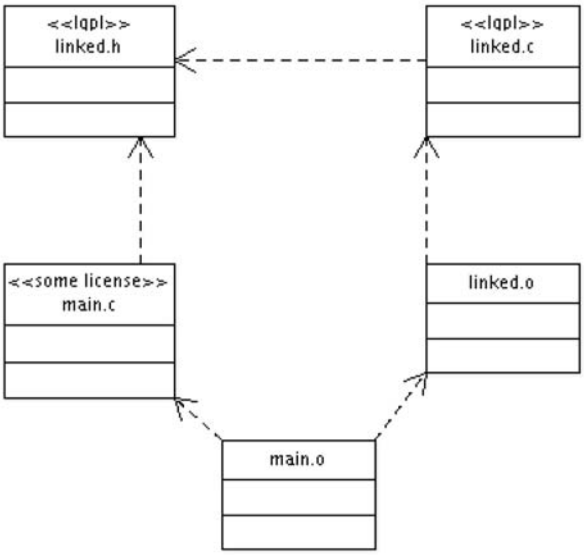


Fig. 3 Linking example illustrated via a class diagram

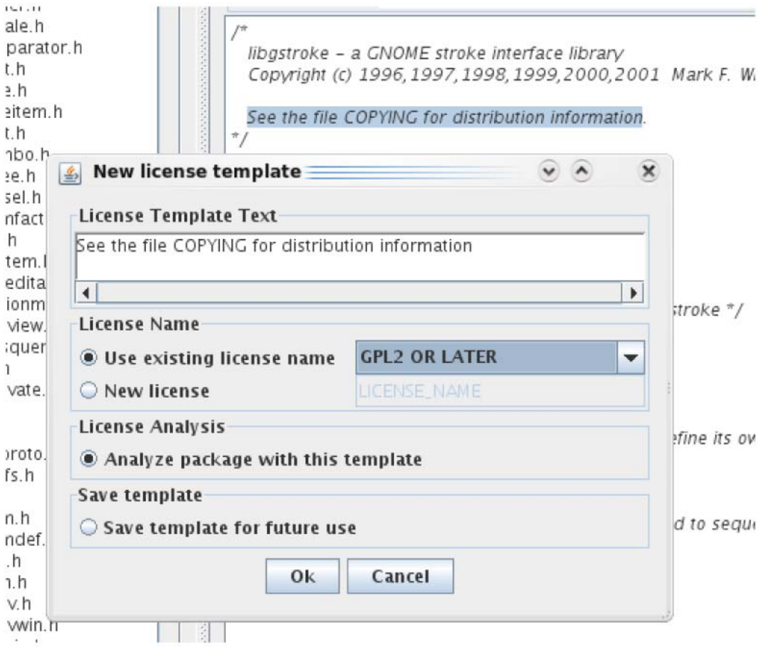
– *main.o* is dependent on *main.c* and *linked.o*

According to the typical license interpretation, in the case of LGPL, static linking is not allowed unless the derivative work (in this case *main.c*) is also licensed under LGPL (or GPL). However, dynamic linking of LGPL components is allowed to a derivative work that can be freely licensed. Thus, if *linked.o* is dynamically linked to *main.o*, ASLA interprets this as compatible linking. If the linking is static, ASLA can report on the incompatibility of the licenses. The results are then displayed to the user, as will be described in Sect. 5.2.8.

5.2.5 User-involved license determination

As mentioned previously, automated license identification cannot be achieved for every file. Because of this, ASLA offers two ways to support manual license determination. The most typical way is to manually set licenses one by one for unidentified source files (F5.1). This is aided by the fact that ASLA lists all source code files that were unidentified in a separated tree entry (see Fig. 6). Another way is to apply a license for a whole module at once (F5.2). There are often situations in which a license file (usually called *COPYING*) is meant to cover all the files in subdirectories, but the source files themselves do not include any reference to the license used (as described in Sect. 3). In these cases the tool user is able to apply this selected license to all the files in the subdirectories.

A typical problem in the license analysis is the inclusion of a library header file that has no license information whatsoever. The files cause ASLA to report on the missing license, even though such libraries are usually under permissive licenses that allow them to be used freely as part of another program. ASLA provides the functionality to exclude a single file or a whole directory from the license analysis—an extremely useful feature in cases of this kind (F5.3).



```
/*
  libgstroke - a GNOME stroke interface library
  Copyright (c) 1996,1997,1998,1999,2000,2001  Mark F. W

  See the file COPYING for distribution information.
*/
```

New license template

License Template Text: See the file COPYING for distribution information

License Name:
[✓ Use existing license name] GPL2 OR LATER
New license LICENSE_NAME

License Analysis:
[✓ Analyze package with this template]

Save template:
Save template for future use

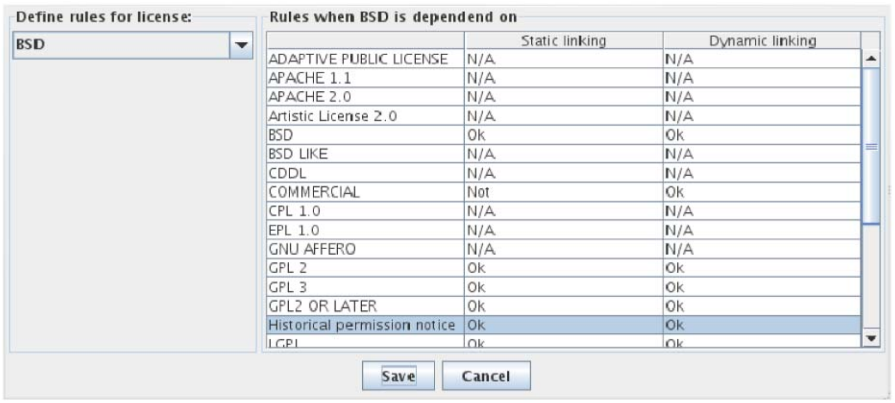
Fig. 4 Creation of a new license template in ASLA

5.2.6 *Addition of new license templates*

To reduce the number of unidentified licenses and the need for manual license identification with other software packages, the user is able to add new license identification templates in two ways. The first way is to manually create a new text file in the directory in which the existing license template files are saved (F6.1). The file format for the new template contains the license name on the first line of the file, and the template text in regular expression form in the following lines. ASLA automatically reads these new files in the next startup.

The second way, i.e. run-time license template addition (F6.2), is especially useful during the license analysis of a package. The tool user is able to select a text in a source file, define a license name for this text and use that information as a license identification template (Fig. 4). In this case ASLA reformats the text into a regular expression and saves it for future use if necessary.

The textbox at the top of the *New license template window* contains the text that the user has selected from the source file. This text can be edited manually if necessary. The user is able either to use an existing license name for the new template or else create a template for a completely new license in the *License name* section of the window. If an existing license name is used, the new template will be a template parallel to the existing templates. The user can choose whether the new template is to be saved for future use by using the last option in the window. Note that if a new license name is defined, the compatibility rules between this new license and existing


Define rules for license:

BSD

Rules when BSD is dependend on

	Static linking	Dynamic linking
ADAPTIVE PUBLIC LICENSE	N/A	N/A
APACHE 1.1	N/A	N/A
APACHE 2.0	N/A	N/A
Artistic License 2.0	N/A	N/A
BSD	Ok	Ok
BSD LIKE	N/A	N/A
CDDL	N/A	N/A
COMMERCIAL	Not	Ok
CPL 1.0	N/A	N/A
EPL 1.0	N/A	N/A
GNU AFFERO	N/A	N/A
GPL 2	Ok	Ok
GPL 3	Ok	Ok
GPL2 OR LATER	Ok	Ok
Historical permission notice	Ok	Ok
LGPL	Ok	Ok

Fig. 5 License compatibility rules in ASLA

licenses cannot be defined automatically. The user must define the compatibility rules for the new license as described below.

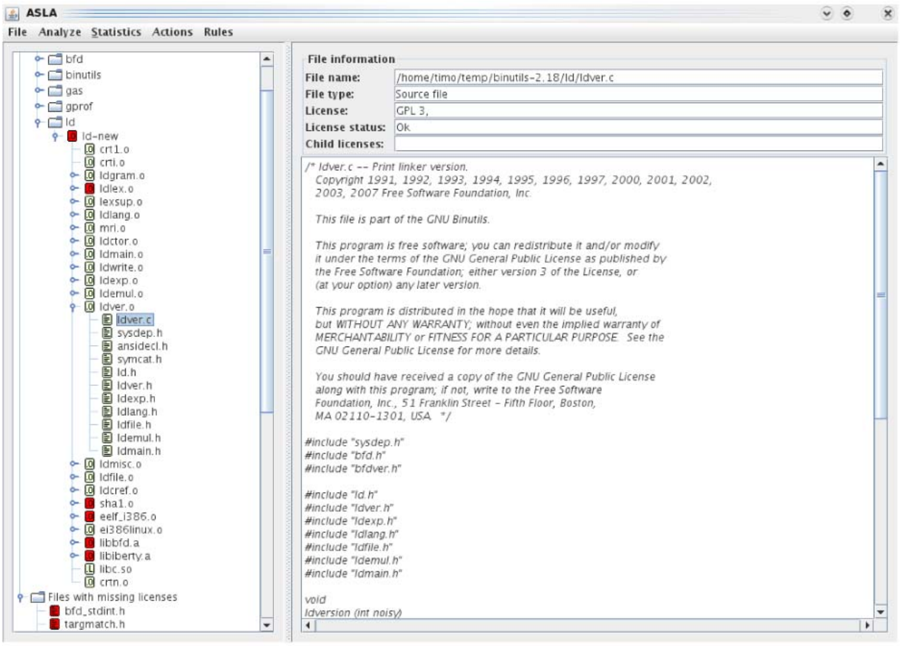
5.2.7 *The visualization and definition of license compatibility rules*

Open source licenses are more complicated than they first appear. As described in Sect. 3, many open source licenses are incompatible with each other, and even lawyers have different interpretations of the licenses and their compatibilities. For this reason, it is possible for the tool user to see and define license compatibility rules according to his/her own interpretations.

ASLA offers a user interface (Fig. 5) for the visualization (F7.1) and definition (F7.2) of compatibility rules. From the drop-down list on the left, the user selects the license for which the rules will be shown, and possibly defined or modified. The first column of the table lists the names of all the licenses supported by ASLA (note that if the user adds a new license identification template with a new license name, it will also be shown in this column). Rule definitions are made for each license separately, the license selected from the drop-down list being the license of the parent file, and the licenses listed in the table being the licenses of the child files. Rules are defined separately for static and dynamic linking (second and third column) for the reasons described in Sects. 3 and 5.2.4.

5.2.8 *Visualization of the results of license analysis*

In most reverse engineering tools one of the main features is the visualization of the information obtained. The ASLA user interface in Fig. 6 has two main sections: the left hand box illustrates the program dependencies by displaying the dependency tree in tree format (F8.1), while the right hand box displays detailed information about selected dependency objects (F8.2). By browsing the tree on the left (F8.3), the user is also able to see the list of all the analyzed objects and their dependencies. The program parts that are successfully analyzed from the license perspective are colored leaf-green, and the objects (leaves) that have some sort of potential license problem



```
bfd
binutils
gas
gprof
ld
  ld-new
    crt1.o
    crti.o
    ldgram.o
    ldlex.o
    lexsup.o
    ldlang.o
    mri.o
    ldctor.o
    ldmain.o
    ldwrite.o
    ldexp.o
    ldemul.o
    ldver.o
      ldver.c
      sysdep.h
      ansidecl.h
      symcat.h
      ld.h
      ldver.h
      ldexp.h
      ldlang.h
      ldfile.h
      ldemul.h
      ldmain.h
    ldmisc.o
    ldfile.o
    ldcref.o
    sha1.o
    eelf_i386.o
    ei386linux.o
    libbfd.a
    libiberty.a
    libc.so
    crtn.o
Files with missing licenses
  bfd_stdint.h
  targmatch.h
```

File information

File name: /home/timo/temp/binutils-2.18/ld/ldver.c
File type: Source file
License: GPL 3,
License status: Ok
Child licenses:

```
/* ldver.c -- Print linker version.
   Copyright 1991, 1992, 1993, 1994, 1995, 1996, 1997, 2000, 2001, 2002,
   2003, 2007 Free Software Foundation, Inc.

   This file is part of the GNU Binutils.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 3 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street - Fifth Floor, Boston,
   MA 02110-1301, USA.  */

#include "sysdep.h"
#include "bfd.h"
#include "bfdver.h"

#include "ld.h"
#include "ldver.h"
#include "ldexp.h"
#include "ldlang.h"
#include "ldfile.h"
#include "ldemul.h"
#include "ldmain.h"

void
ldversion (int noisy)
```

Fig. 6 ASLA main view

are colored red. An identified license problem will involve either an unrecognized license or the use of incompatible licenses.

The visualized information is useful for both the full and the partial reuse of software packages. For partial reuse, the different leaves of the tree indicate potential reusable components. In the case of full reuse, the tree format introduces the dependencies of the different parts of the software and indicates how the licenses of the compiler outputs are collected from the source files. By selecting an object in the tree, the tool user is provided with the following detailed information concerning the object in question: *full file name, file type, license of the file, license status* (OK, unrecognized license, unrecognized child licenses, incompatible licenses), *list of licenses found from child objects*, and (in the case of a source file) the *actual source code*. The license status is automatically-formed state information concerning the results of the license analysis. If the license status is OK, the license of the file and the licenses of all its child objects have been successfully identified, and are seen to be compatible with each other. The license statuses *unrecognized license* and *unrecognized child licenses* indicate that the licenses of some files have not been successfully identified. The license status *incompatible licenses* indicates that all the licenses have been successfully identified, but that the licenses are incompatible with each other as defined by the compatibility rules.

5.2.9 Statistics and summary information

During the license analysis, ASLA collects statistical information for the user. When a source package is analyzed by ASLA, it provides statistical and summary information about the licenses found and the files used from the source package (F9.1). ASLA lists all the licenses applied, the number of files with each applied license, and the number of files whose licenses were unrecognized. The statistics also include information about the number of files from the source package and the number of external files included in the final binary output.

6 Tool evaluation

The approach that we have described in this paper offers an extendable license analyzer for programmers interested in using and reusing software packages. The approach is designed to be used in the analysis of existing software packages. An especially rich resource of possibly reusable software is to be found in OSS packages that can be downloaded from the Internet and compiled from source codes. The user needs for automated license analysis were presented in Sect. 4 and the functionality of ASLA in Sect. 5. In this section, we shall evaluate the main system-level features based on the non-functional user needs stated in Sect. 4.3.

The main high-level (i.e. system level) feature of the tool is dependency and license analysis (F1.1). The second high-level feature, i.e. the license identification of source code files (F1.2), is designed for use with packages for which dependency analysis cannot be conducted using our approach. It expands the usability of our tool (in terms of license identification) to packages that could not be analyzed otherwise.

As described in Sect. 3 some open source licenses are incompatible with each other, and are especially incompatible with proprietary licenses. Since we are able to identify the licenses of source files and the dependencies between build process outputs, we can automatically identify possible license compatibility problems (F4.1) in software components, using the compatibility rules (F4.2). This automated compatibility analysis is of great utility in many cases, since there are many OSS packages that contain hundreds or thousands of source files with different licenses.

The key result of the license analysis process is the presentation of all the required information to the user. In addition to displaying information, the license analysis often includes the users' active participation. The typical license analysis process reveals several problems in an initial analysis. The ASLA approach enhances the flexibility and accuracy of license identification by offering support for manual license determination (F5.1 and F5.2) and exclusion of unnecessary files (F5.3). ASLA also offers the possibility of adding new license templates (F6.1 and F6.2) that can be used for license identification. These features help the user to further analyze the software package, and to separate unnecessary incompatibility reports from those that need further analysis or contain actual incompatibilities.

As described above, ASLA provides the functionality needed to perform license analysis. To measure the efficiency (N13) and coverage (N14) of our approach, we selected and analyzed 12 open source packages using the ASLA tool. The test was conducted in Kubuntu Linux (version 8.10) with 1 GB of memory and a 2.2 GHz processor. Our goal was to select packages of different sizes, different application areas, and programming languages, in order to test the analyzer with a wide variety of packages. The main characteristics of these packages are presented in Table 3.

Table 3 OSS packages analyzed

Package	Application type	Implementation languages	KLOC
AFPL Ghostscript (AFPL Ghostscript 2008) (ghostscript-8.54)	Postscript and PDF interpreter, converter, and application library	C, Postscript	673
Apache HTTP Server (Apache 2008) (httpd-2.2.6)	Web server	C	316
Azureus (Azureus 2008) (Azureus_3.0.4.2)	File-sharing application	Java	25
Bugzilla (Bugzilla 2008) (bugzilla-3.0.3)	Bug tracking application	HTML, Perl	162
GIMP—The GNU Image Manipulation Program (GIMP 2008) (gimp-2.4.0)	Image processing application	C	863
GNU Binutils (GNU Binutils 2007) (binutils-2.18.50)	System binary tools	C	1289
GNU Go (GNU Go 2008) (gnugo-3.6)	Board game	C	134
gnuplot (gnuplot 2008) (gnuplot-4.2.2)	Mathematical visualization software	Assembler, C++	736
JBoss (JBoss 2008) (jboss-4.2.2.GA)	J2EE application server	Java	177
Mozilla Firefox (Mozilla 2008) (firefox-2.0.0.9)	Web browser	Assembler, C++, C, Java, Perl, Python	2063
Pidgin IM client (Pidgin 2008) (pidgin-2.5.4)	Instant messaging client	C, Perl, Python	332
Subversion (Subversion 2008) (subversion-1.4.6)	Version control tool	C, C++, Java, Perl, Python	627

The column *Package* gives the name and version of the source code package. The second column, *Application type* describes the purpose of the software, i.e. its main functionality. The third column lists the *Implementation languages* that have been used in the package. The last column, *KLOC*, describes the size of the source package in thousands of lines of code.

The results are displayed in Table 4. We focus on evaluating the non-functional needs related to the two main system-level features. The first four columns (after the Package name column) represent the results of license identification of the source code files (F1.2), i.e. the identification of the license of each source code file. The initial results are combined with the results of re-analysis after user-involved license determination (F5.3) and the addition of new license templates (F6.2). The last five columns describe the results of dependency and license analysis (F1.1). This analysis covers all basic user needs and conducts a thorough license analysis for the input package, as described in Sect. 5.2. Furthermore, in this case, the initial results are combined with the results of re-analysis after the addition of new license templates (F6.2). To summarize, whereas dependency and license analysis excludes files from

Table 4 Efficiency and coverage test results

	License identification of source code files (F1.2)				Dependency and license analysis (F1.1)				
Package name	NF_TOT	NFI_PLI	NFI_PLI_F	EFF_PLI	NF_TES + NF_EXT	NF_DIF	NFI_DLA	NFI_DLA_F	EFF_DLA
AFPL Ghostscript	1568	10	1490	56 s	1076	61	10	1037	67 s
		(1%)	(95%)	(28 files/s)	(1076 + 0)		(1%)	(96%)	(16 files/s)
Apache HTTP Server	821	21	740	24 s	327	15	17	321	13 s
		(3%)	(90%)	(34 files/s)	(327 + 0)		(5%)	(98%)	(25 files/s)
Azureus	2713	1522	2276	20 s	N/A	N/A	N/A	N/A	N/A
		(56%)	(84%)	(136 files/s)					
Bugzilla	608	458	173	1 s	N/A	N/A	N/A	N/A	N/A
		(75%)	(95%)	(182 files/s)					
GIMP	2732	2319	2347	135 s	3146	1424	2776	2790	325 s
		(85%)	(86%)	(19 files/s)	(2416 + 630)		(91%)	(92%)	(9 files/s)
GNU Binutils	1533	1222	1303	13 s	398	10	352	368	18 s
		(80%)	(95%)	(118 files/s)	(398 + 0)		(88%)	(92%)	(22 files/s)
GNU Go	120	6	118	135 s	165	93	57	151	124 s
		(5%)	(98%)	(1 files/s)	(103 + 62)		(35%)	(92%)	(1 files/s)
gnuplot	183	1	145	5 s	218	58	90	186	11 s
		(1%)	(79%)	(36 files/s)	(107 + 111)		(41%)	(85%)	(20 files/s)
JBoss	8499	8250	8261	18 s	N/A	N/A	N/A	N/A	N/A
		(97%)	(97%)	(472 files/s)					
Mozilla Firefox	17 823	14321	14442	85 s	6543	2532	6225	6370	272 s
		(80%)	(81%)	(225 files/s)	(5978 + 565)		(96%)	(98%)	(24 files/s)
Pidgin IM client	955	784	860	20 s	1395	527	1201	1264	216 s
		(82%)	(90%)	(48 files/s)	(665 + 730)		(86%)	(90%)	(6 files/s)
Subversion	661	1	525	16 s	312	27	37	312	21 s
		(0%)	(79%)	(37 files/s)	(276 + 36)		(12%)	(100%)	(15 files/s)

the license identification, the license identification of the source code files attempts to identify the license of every source file found in the package.

NF_TOT (Number of files—total) Total number of source files found in a package.

NFI_PLI (Number of files identified—license identification of source code files (F1.2)) Number of source files with an identified license in the license identification of source code files, initially. This means the number of files that are identified by ASLA without any additional user involvement.

NFI_PLI_F (Number of files identified in the license identification of source code files—licenses identified finally) Number of source files with identified license in the license identification of source code files after participation of the user (features F5.3 and F6.2) and re-run of the analysis.

EFF_PLI (Efficiency—license identification of source code files) Total time spent on license identification of source code files. This analysis includes all source files found in the package analyzed (i.e. number of files shown by NF_TOT).

NF_TES + NF_EXT (Number of files—tested and Number of files—external) Number of source files used from the package in our test environment, plus the number of "external" files, i.e. files included from system libraries. This list of files is gathered during the dependency analysis, in which files that are used from the package in our execution environment are separated from the unused files (see feature F2.2).

NF_DIF (Number of files—dependency information files) Number of dependency information files that have been created during the build process and have been read and analyzed by ASLA during the dependency and license analysis (F1.1).

NFI_DLA (Number of files identified—dependency and license analysis) Number of source files with an identified license in dependency and license analysis initially, i.e. number of files that are identified by ASLA without any additional user involvement.

NFI_DLA_F (Number of files identified in dependency and license analysis—licenses identified finally) Number of source files with an identified license in dependency and license analysis (F1.1) after participation of the user (F6.2) and re-run of the analysis.

EFF_DLA (Efficiency—dependency analysis and license analysis) Total time spent on dependency and license analysis (F1.1). The number of files analyzed is NF_TES + NF_EXT + NF_DIF.

The results are divided into dependency and license analysis, and the license identifications of source code files. Azureus, Bugzilla, and JBoss were implemented using programming languages that cannot be compiled using GCC, so only license identification of source code files was conducted. Hence these packages have N/A (not available) in NF_TES + NF_EXT, NF_DIF, NF_DLA, NF_DLA_F, and EFF_DLA columns in Table 4.

In the case of dependency and license analysis, we read and analyzed from 10 to 2532 dependency information files per open source package. We were able to identify licenses in 1%–97% of the source files listed in those files, without any user involvement. These uneven identification results are due to the fact that some of the source packages analyzed had their own unique way of indicating the license used in the source files. The results were greatly improved by applying new license templates. The technique described in Sect. 5.2.6 (feature F6.2) was used in all cases. We were able to identify licenses in 75%–100% of the files by applying 1–5 new templates to each package. For example, in the case of gnuplot, where we were able to identify the license of only 41% of the source files in the initial analysis, two license identification templates were added, and the success rate increased to 85%. The source files that remained unidentified had no indication whatsoever of the license used. In other packages, too, most of the instances of non-identification were caused by this same reason. Note that when applying the new license templates, the tool user has to make sure that the licensing rules for these new templates are also created (see Sect. 5.2.7), unless the new template is a variant of a previously known license. This ensures that the automated dependency analysis will also cover the new licenses.

In the case of the license identification of source code files, we were able to identify the license in 1%–97% of the source files without user involvement. Because the license identification of source code files is a procedure which attempts to identify the license of every single file that can be interpreted as a source file, we also seek to identify files which are, most probably, not supposed to be regarded as source files (e.g. HTML help files), and which therefore have no license identification whatsoever. This affects the success rate of the identification. By applying new license identification templates (F6.2), and by excluding unnecessary files (F5.3), the identification results improved to between 79% and 98%.

The efficiency of the license identification of source code files (feature F1.2) is between 1 and 472 files per second. This large variance can mostly be explained by two factors: (1) Successful identification is usually very efficient (some milliseconds/file), especially when most files within the package have the same license. (2) In the case of slow results, the license identification for most files is unsuccessful. In these cases *java.util.regex* classes (Java's standard regular expression library) uses up most of the time. Unsuccessful regular expression results constitute one, but not the only explanatory factor, since—depending on the file contents—regular expressions can fail very quickly, or can take quite a long time. We presume that this variation is due to the implementation of *java.util.regex*.

Dependency analysis (features F2.1 and F2.2) and compatibility checking (feature F4.1) also affect the total efficiency. In most cases the overhead produced by these features is about 1.5 to 2 times the time spent on license identification (AFPL, Apache, GIMP, gnuplot, Pidgin, and Subversion), but in some cases this figure can rise more than eight-fold (Mozilla). In any case, compared to manual analysis of the packages, these results are very good.

The high efficiency of ASLA as compared to manual analysis is based on a number of features: the existence of identification templates and reuse of the user's manual license identification (addition of license templates). These aspects ensure that once the identification of certain text as belonging to some specific license has been achieved, the identification can be reused efficiently. By using ASLA, the user is able to focus on solving the license issues in a small subset of files in the package, since the license of most of the files is recognized automatically. To further enhance the efficiency of the required manual analysis, the analyzer displays the unsuccessful license identification results separately.

The evaluation conducted has provided two main results: (1) the ASLA tool enables the analysis of large OSS programs in a reasonable time, and (2) it provides information which can be valuable to software development teams when reusing OSS-licensed software. The tool is also easily extendable, and it is not restricted to any particular programming language.

7 Comparison of license analyzers

Over the past years there have been some attempts to create systems that automatically identify licenses and check software license compatibility. In 2001 Metrowerks introduced the 'GPL compliance toolset', which was 'built to guide embedded systems developers through the extensive maze of software code license compliance issues' (GPLToolkit 2007). Also in 2001, LIDESC (2007), which is an awareness tool for preventing unintended violations of license terms, was introduced. The Open Source License Checker (OSLC) (OSLC 2007), which identifies licenses from source packages and reports them to the user, was introduced in 2006. The OSLC has not been described in academic papers, but because it is open source, its functionality can nevertheless be studied in detail.

LIDESC, OSLC and ASLA are tools that are designed for software package license analysis on a local computer. There are also server based, so-called repository tools, which store the information of the license analysis in a repository. The repository can be browsed for the results of the analysis. Two companies, Black Duck (2009) and Palamida (2009), provide extensively used commercial open source analysis tools of this kind. Both these companies provide a tool set for identifying the licenses used, for validating software contents and for verifying license compliance. There is also an open source tool called FOSSology (2009) which provides similar functionality.

7.1 Tool comparison

We compared three license analysis tools: ASLA (version 1.1), OSLC (version 2.0), and FOSSology (version 1.0). The comparison is based on the user needs described in Sect. 4. In its relevant parts, Pidgin (version 2.5.4) is used as an example software package to illustrate the functionalities of the tools. We selected Pidgin as an example OSS for a number of reasons, including the fact that Pidgin is a popular OSS for its purpose (i.e. instant messaging). It is also an adequate example in the sense that it includes files with different licenses and includes several potentially reusable components. It is clearly a non-trivial system (421 KLOCs, including a total of 3750 files, of which 955 are source code files). Pidgin could be reused e.g. by reimplementing the user interface in a new usage context (e.g. a mobile device) or by reusing its various instant messaging protocol implementations.

We excluded the other tools from the comparison for the following reasons. In the case of the 'GPL compliance toolset', there is no up-to-date information on the product, so it can be assumed that the tool no longer exists. LIDESC was also excluded from the analysis, since there was no working version available (the version we obtained from the Internet did not compile). Palamida and Black Duck were not included in the comparison because: (1) we were not able to obtain working copies of the software, (2) there is no detailed published information on these tools, and (3) according to HP (2008), FOSSology provides similar functionality.

7.1.1 Basic user needs

From the license analysis point of view, ASLA, OSLC, and FOSSology provide fairly similar functionality for fulfilling the basic user needs. All these tools identify the contents of the packages and attempt to identify the licenses of files within the packages. ASLA and OSLC have license compatibility rules, and they conduct compatibility checking. After the analysis, all these tools visualize the results for the user. However, the features included, and their implementation, differ as described below.

In their license analysis process, Nordquist et al. (2003) describe the identification of the program modules that are included in a program in a particular environment (N1) as creation of the bill of materials (BOM). This user need is fulfilled only by ASLA. Because ASLA dependency and license analysis is based on separation of used and unused source files, ASLA does not attempt to identify the licenses of all source code files when conducting its analysis. It concentrates on the parts of the software that are actually used in a specific environment, and excludes all other files. OSLC and FOSSology extract the packet to be analyzed and treat the extracted structure (directories and files in those directories) as their BOM, which is a basic level internal data structure (as part of L1 in Sect. 2.3). This approach is also available in ASLA by using feature F1.2. The differences between these approaches will become evident with reference to the ways in which the tools visualize their results to the end user (see below).

License identification of each source code file (N2) is a need fulfilled by all the tools in the comparison. However, the implementations differ in terms of flexibility and efficiency. Furthermore, the number of the identification templates provided differs greatly: FOSSology has almost 300 templates, whereas ASLA has 37 and OSLC 43. The ASLA approach of using regular expressions has been described in Sect. 5.2.3. OSLC identifies the licenses of each source file by using a technique introduced by Heckel (1978). OSLC reports matching results in percentages indicating how closely the text found in a source file matches the predefined template (e.g. a X% match to GPL2). The OSLC identification technique makes it possible to identify licenses even if the exact phrase or license text is not found in the source code file. It also solves the problem of occurring different comment and white space characters, as described in Sect. 5.2.3. The license matching technique of FOSSology combines the techniques used in ASLA and OSLC. It offers the possibility to define white spaces or undefined words in the middle of search templates (like the regexp-based templates used by ASLA) but reports the matches in percentages (like OSLC). This technique is flexible and accurate, but fairly slow as compared to the license identification of ASLA and OSLC.

In order to compare the differences in license identification (N2), we ran license analysis on each tool using the Pidgin package. Table 5 describes the results of this comparison. The column *Files analyzed* lists the number of files analyzed by each

Table 5 License identification results for the example case of Pidgin 2.5.4

Tool	Files analyzed	Licenses identified	Count	Time required for analysis
ASLA	955			20 s
		GPL	662	
		LGPL	67	
		MIT	52	
		All rights reserved	24	
		BSD	2	
		Public domain	2	
OSLC	943			15 s
		GPL	700	
		LGPL	66	
		All rights reserved	24	
		BSD	2	
FOSSology	3750			2 hours 33 min
		GPL	780	
		LGPL	148	
		Phrase	101	
		FSF style	54	
		BSD	2	
		Public domain	2	

tool. FOSSology analyzes every file found in the package (3750 files) whereas OSCL (943 files) and ASLA (using feature F1.2) (955 files) only analyze files that are considered to be source code files (excluding e.g. installation scripts and Makefiles). The column *Licenses identified* lists the names of the licenses identified, and column *Count* describes the number of identifications for that license. The column *Time required for analysis* shows the amount of time that the tool spent on the license identification. It can be seen that ASLA and OSLC perform the analysis in a matter of seconds, whereas FOSSology spent more than two and a half hours. FOSSology identifies some licenses that are not identified by OSLC and ASLA. However, these identifications (*FSF style* and *Phrase*) are found mostly in Makefiles and configure scripts that are discarded by ASLA and OSLC. In addition, some of the FOSSology license identifications (such as number of *Phrase* identifications and some other individual identifications) were false positives, in the sense that they did not actually refer to any license. These are excluded from the Table 5, since including them would give the erroneous impression that FOSSology identifies licenses in our example case that ASLA or OSLC do not.

Dependency identification of program modules (N3) is implemented in both the ASLA and OSLC tools. The creation of the dependency map in ASLA is based on both file and binary dependencies. OSLC only provides dependency analysis on a source code level. Dependency identification in OSLC is based on the lines that define what files are included in the source file (e.g. by using import in Java and #include in C/C++). Due to this implementation technique, in the case of e.g. C/C++, OSLC is not able to find system files that are included, since it will not be able to find the directory containing the files. FOSSology does not provide creation of the dependency map at all.

The formation of license compatibility rules (N4) and the identification of licensing problems (N5) are provided by both ASLA and OSLC. OSLC includes rules between licenses, and it recognizes licensing conflicts between and within the source files much as ASLA does. An example of the identification of licensing problems is provided below.

Traditionally, reverse engineering tools visualize their results to the end user in some form. However, due to the nature of the license analysis task, the visualization of the results (N6) is fairly simple in all the tools compared here. ASLA visualizes the dependency map (as described in Sect. 5.2.8), and OSLC and FOSSology visualize the top directory of the analyzed package.

For license analysis, it is more natural to make queries and browse the results than to attempt to visualize them all at once. Each of the tools offers this kind of functionality as described in more detail below.

7.1.2 Interaction related user needs

All these tools enable browsing or requests for results (N7) after a package has been analyzed. If we look at our example case of Pidgin, the tools provide the following information.

As we browse the dependency map created in the ASLA analysis, we can discover that Pidgin compilation creates the Pidgin main program and over 30 shared objects. A more detailed analysis reveals that these shared objects are dynamically loaded by Pidgin during the runtime. This information is not available in OSLC or FOSSology. ASLA provides information which shows that all compiled binary files have some potential licensing problems. In most cases, some files are included whose license has not been identified. For example libirc.so (Internet relay chat implementation) includes 15 system header files which have no licensing information in them. Missing license files are also presented to the user in OSLC. OSLC provides different ways to filter or browse the information about the package analyzed. When using OSLC, we can choose to filter only those files that have a missing license, files that have an identified license, files that have an uncertain license, and files that have licensing conflicts. In addition to missing licenses, both ASLA and OSLC inform the user of a licensing conflict in the Pidgin package: some files (in a shared object libnovell.so as identified by ASLA) include the phrase *All rights reserved*. This phrase is incompatible with all open source licenses, since it excludes all rights from the user (see Sect. 3 for the rights that open source licenses must provide). However, all the files that include the *All rights reserved* phrase also include information to the effect that these files are licensed under GPL v2 license. What this contradictory information actually means is something that will ultimately be for lawyers to decide. This is a typical example of how ASLA and OSLC identify potential licensing problems and present them to the tool user. Browsing the Pidgin package using FOSSology provides little information besides what is described in Table 5. The tool user can either browse the directories and files of the package or list the licenses that are found in the package. License names are links that lead the user to a list of files that are identified under the indicated license. When opening a single file, FOSSology provides a view that highlights the part of each file that is identified as a license match.

Manual source code license determination (N8) is provided only by ASLA. For example, a closer inspection of unidentified files in libirc.so reveals that these files are licensed under a permissive license so they can be used as part of a program using any selected license. Either by determining their licenses manually or by excluding these files from the license analysis (as described in Sect. 5.2.5), we can discover that libirc.so does not have any potential licensing problems. This information itself is useful to the user, since following the exclusion we have found a potentially reusable component, one that is licensed under GPL v2 and has no internal licensing problems. Manual license identification is an essential part of the ASLA license analysis process, since no analyzed package can obtain OK status (described in Sect. 5.2.4) unless the licenses of all the source code files are identified or the source files are excluded from the analysis. Developers of other tools do not seem to consider this to be a problem, since the lack of a license for a file does not produce any compatibility warnings in OSLC, and gives no indication whatsoever in FOSSology. In addition, OSLC does not provide any features that would allow the tool user to exclude the conflicting files (see the case above), or to identify files as being licensed under some known license.

The need for a new license template addition (N9) also comes up when we analyze our example package (Pidgin). There are several files that have the following phrase: *For copying and distribution information, see the file "mit-copyright.h"*. When we examine the file mit-copyright.h we discover that it contains a standard MIT license. Using ASLA, the user can either add the identification template during the license analysis or else manually add a new template as described in Sect. 5.2.6. In the case of OSLC, we require a text file describing the template and a meta file describing the compatibility rules with other files in order to add the new template to the OSLC. If we wished to add the same analysis template to FOSSology, we would need to rebuild the entire software and modify the existing database. It should be noted that FOSSology also provides a possibility for grouping licenses. For example, the phrases *GNU General Public License version 2* and *GPL version 2* may both be parts of the 'GPLv2' class. This feature can be used for identifying and grouping licenses. By adding a new license template for the phrase above, we can identify the license of 51 previously unidentified source code files in Pidgin package.

Visualization of the license compatibility rules (N10) is provided only by ASLA. The definition of the rules (N11) can also be found from OSLC.

Statistics and summary information on license analysis (N12) is implemented in all the tools compared. In all cases, a list is displayed of the names of the licenses used within the package, and a count of the files that use each of these licenses. In the case of OSLC the following information is also presented to the tool user after the analysis is complete (an example from the Pidgin analysis): Number of files: 3750. Number of source files: 943. Distinct licenses: 7. Conflicts (reference 86 and global 5).

FOSSology provides some features that are not present in either ASLA or OSLC. The results produced are saved for future use, and the tool user can browse the results of different packages. Packages can be uploaded to the repository of FOSSology

Table 6 Coverage of the compared tools in terms of meeting user needs

	ASLA	OSLC	FOSSology
N1 Identification of the program modules that are included in the program within a particular environment.	Yes	Partial	Partial
N2 License identification of each source file.	Yes	Yes	Yes
N3 Dependency identification of program modules.	Yes	Partial	No
N4 The formation of license compatibility rules.	Yes	Yes	No
N5 The identification of licensing problems.	Yes	Yes	No
N6 The visualization of license analysis results.	Yes	Yes	Yes
N7 Browsing of the results of the license analysis.	Yes	Yes	Yes
N8 Manual determination of the source code license.	Yes	No	No
N9 The addition of license identification templates.	Yes	Yes	Yes
N10 The visualization of license compatibility rules.	Yes	No	No
N11 The definition of license compatibility rules.	Yes	Yes	No
N12 Statistical information on the analyzed software package and the license analysis.	Yes	Yes	Yes

using various methods. FOSSology also provides the functionality to manage the repository (e.g. creation and management of upload folders, management of uploaded packages and management of license terms and groups). Furthermore, the jobs (i.e. different analysis tasks) can be managed via this tool.

7.1.3 Tool comparison: summary

All the tools compared identify the licenses of source code files, with fairly similar results concerning the licenses identified (see Table 5). However, the FOSSology approach to license analysis differs from that of ASLA or OSLC. Whereas ASLA and OSLC try to present potential licensing problems to the tool user, FOSSology concentrates more on conducting detailed license identification and saving the results for future use. Table 6 lists the available features of the compared tools regarding the user needs. *Yes* means that the tool provides features fulfilling a user need, while *No* means that the tool has no functionality for fulfilling that user need. In some cases OSLC and FOSSology provide functionality that partially fulfills the user need (*Partial*). OSLC and FOSSology identify the program modules (N1), but they do not exclude unused files. OSLC identifies the dependencies (N3) only partially (at source code level) as described above, whereas FOSSology does not include dependency analysis at all. FOSSology does not provide any functionality for the formation of license compatibility rules (N4) or for the identification of licensing problems (N5). Manual source code license determination (N8) is lacking in OSLC and FOSSology. Even though OSLC provides definitions of license compatibility rules (N11), it lacks the visualization of the rules (N10). FOSSology has no features to fulfill either of these needs.

As indicated above, ASLA offers more user interaction-related features for license analysis than the other compared tools, and it also offers more detailed license compatibility analysis. This is mainly due to the fact that ASLA bases its analysis on dependency information in such a way that the information is combined with license compatibility rules, and is thereby more comprehensive than the information used in the other tools. OSLC bases its license compatibility analysis on operations at source code level, while FOSSology does not provide any actual compatibility analysis. ASLA also combines the required external files as part of its license analysis, whereas the other tools analyze only the given package and do not take external dependencies (and their licenses) into account.

8 Conclusions and summary

This paper has presented a reverse engineering approach and its implementation via a tool termed ASLA, the purpose being to retrieve software license information from source code modules. The retrieval, comprehension and determination of licenses is important for the effective reuse of software components. ASLA has been motivated by a typical problem in OSS development: license information should be gathered, but reliable and detailed manual license information retrieval is slow and laborious. The paper has described ASLA's user needs, system architecture, tool features, and tool evaluation.

In reflecting on our experiences, we can make a number of observations regarding the support that can be provided for dealing with software license information (these relate to the information-retrieval support levels set out in a publication by Koskinen et al. 2004). First of all, license analysis is a new kind of information need, in the sense that previous empirical studies on information needs of software maintainers have not paid attention on it. Secondly, the dependencies between the licenses of parent and child files complicate the analysis, creating a need to provide license compatibility information—which is in fact a new specific information requirement. Thirdly, the classification applied here is able to include all the features required. Fourthly, ASLA provides support at each level, although support at levels L3–L5 has currently intentionally been left on a rather modest level. The main reason to this design choice is that no empirical information has been available concerning whether more sophisticated features at these levels would actually provide added value for potential tool users. Because of this, only basic-level functionalities have been implemented for those levels.

This paper has extended the previous work (Tuunanen et al. 2006) by describing the architecture and functionality of ASLA in a more detailed manner, and especially by providing an extended evaluation of the tool. The evaluation shows that ASLA performs well in its task of identifying licenses and identifying dependencies between program parts. License identification coverage is good, and manual identification is flexible. The dependency and license information gathered offers us the possibility to create license rules that will inform tool users of potential licensing problems. It would be extremely interesting if automated license analysis could draw deeper IPR conclusions based on the license information obtained. However, this is not possible due to differing license interpretations, and to the possible legal consequences that we, as the tool provider, might face if we strayed into the realm of giving legal advice without having the necessary qualifications to do so.

It can be concluded that ASLA addresses an important problem and provides promising results in terms of coverage and efficiency in identifying OSS licenses. The importance of performing empirical studies on reuse has been emphasized by Frakes and Kang (2005). We plan to conduct further empirical studies in future regarding three aspects: (1) discovering the information needs of software experts related to dealing with OSS licenses, (2) measurement of the suitability of ASLA for satisfying typical information needs, and (3) investigation of the evolution of relevant characteristics such as reusability, change-rate, and maintainability of OSSs based on public repositories.

References

AFPL Ghostscript. http://pages.cs.wisc.edu/~ghost/doc/AFPL/index.htm (2008). Accessed 11 August 2008

Aho, A., Sethi, R., Ullman, J.: Compilers—Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)

Anderson, P., Reps, T., Teitelbaum, T.: Design and implementation of a fine-grained software inspection tool. IEEE Trans. Softw. Eng. **29**(8), 721–733 (2003)

Andritsos, P., Miller, R.: Reverse engineering meets data analysis. In: Jacobs, A. (ed.) Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001), pp. 157–166. IEEE Computer Society, Los Alamitos (2001)

Apache. Apache HTTP Server Project. http://httpd.apache.org/ (2005). Accessed 12 August 2008

Apache License, Version 2.0. http://www.opensource.org/licenses/apache2.0.php (2007). Accessed 7 October 2007

Azureus. http://azureus.sourceforge.net/ (2008). Accessed 11 August 2008

Bellay, B., Gall, H.: An evaluation of reverse engineering tool capabilities. J. Softw. Maint. **10**(5), 305–331 (1998)

Black Duck. http://www.blackducksoftware.com/ (2009). Accessed 13 January 2009

BSD, The BSD License. http://www.opensource.org/licenses/bsd-license.php (2007). Accessed 7 October 2007

Bugzilla. http://www.bugzilla.org/ (2008). Accessed 11 August 2008

Capiluppi, A., Lago, P., Morisio, M.: Characteristics of open source software projects. In: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR 2003), pp. 317–330. IEEE Computer Society, Los Alamitos (2003)

Cross, J., Chikofsky, J., May, J.: Reverse engineering. Adv. Comput. **35**, 199–254 (1992)

Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M., Succi, G.: Preface. In: Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M., Succi, G. (eds.) Open Source Systems. IFIP International Federation for Information Processing, vol. 203, pp. V–VI. Springer, Berlin (2006)

de Jonge, M.: Build-level components. IEEE Trans. Softw. Eng. **31**(7), 588–600 (2005)

FOSSology. http://fossology.org/ (2009). Accessed 13 January 2009

Frakes, W., Kang, K.: Software reuse research: status and future. IEEE Trans. Softw. Eng. **31**(7), 529–536 (2005)

Frakes, W., Terry, C.: Software reuse: Metrics and models. ACM Comput. Surv. **28**(2), 415–435 (1996)

Free software definition. http://www.fsf.org/licensing/essays/free-sw.html (2003). Accessed 21 November 2007

Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch: Why reuse is so hard. IEEE Softw. **12**(6), 17–26 (1995)

GCC, GNU Compiler Collection. http://gcc.gnu.org (2007). Accessed 13 September 2007

GIMP, GNU image manipulation program. http://www.gimp.org/ (2008). Accessed 11 August 2008

GNU Binutils. http://www.gnu.org/software/binutils/ (2007). Accessed 13 September 2007

GNU Go. http://www.gnu.org/software/gnugo/ (2008). Accessed 11 August 2008

GNU Libtool. http://www.gnu.org/software/libtool/ (2008). Accessed 5 May 2008

gnuplot. http://www.gnuplot.info/ (2008). Accessed 11 August 2008

GPLToolkit. http://www.prnewswire.com/cgi-bin/stories.pl?ACCT=104&STORY=/www/story/08-28-2001/0001562502&EDATE= (2007). Accessed 21 November 2007

Heckel, P.: A technique for isolating differences between files. Commun. ACM **21**(4), 264–268 (1978)

HP may accidentally kill Black Duck & Palamida. http://hp.sys-con.com/node/490782 (2008). Accessed 11 January 2009

JBoss application server. http://www.jboss.org/ (2008). Accessed 11 August 2008

Koskinen, J.: Automated transient hypertext support for software maintenance. Jyväskylä Studies in Computing 4, Ph.D. thesis, University of Jyväskylä, Jyväskylä, Finland (2000)

Koskinen, J., Lehmonen, T.: Analysis of ten reverse engineering tools. In: International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE) (2008, in press)

Koskinen, J., Salminen, A., Paakki, J.: Hypertext support for the information needs of software maintainers. J. Softw. Maint. Evol.: Res. Pract. **16**(3), 187–215 (2004)

Lehman, M., Perry, D., Ramil, J.: Implications of evolution metrics on software maintenance. In: Proceedings of the International Conference on Software Maintenance (ICSM 1998), pp. 208–217. IEEE Computer Society, Los Alamitos (1998)

Letovsky, S., Soloway, E.: Delocalized plans and program comprehension. IEEE Softw. **3**(3), 41–49 (1986)

LIDESC: Librock License Awareness System. http://www.mibsoftware.com/librock/lidesc/ (2007). Accessed 22 November 2007

Lientz, B., Swanson, E., Tompkins, G.: Characteristics of application software maintenance. Commun. ACM **21**(6), 466–471 (1978)

Lindvall, M.: Impact analysis in software evolution. Adv. Comput. **59**, 130–211 (2003)

Microsoft: Microsoft software license terms. Microsoft developer network (MSDN) subscription operating systems, professional, and premium editions. http://msdnaa.fei.stuba.sk/MSDN%20EULA%20Eng.pdf (2007). Accessed 1 July 2008

MIT: The MIT License. http://www.opensource.org/licenses/mit-license.php (2007). Accessed 7 October 2007

Mockus, A., Fielding, R., Herbsleb, J.: Two case studies of open source software development: Apache and Mozilla. ACM Trans. Softw. Eng. Methodol. **11**(3), 309–346 (2002)

Morad, S., Kuflik, T.: Conventional and open source software reuse at Orbotech—an industrial experience. In: Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering, pp. 110–117. IEEE Computer Society, Los Alamitos (2005)

Morisio, M.: Success and failure factors in software reuse. IEEE Trans. Softw. Eng. **28**(4), 340–357 (2002)

Mozilla Firefox. http://www.mozilla.org/firefox/ (2008). Accessed 3 July 2008

MPL, Mozilla Public License 1.1 (MPL 1.1). http://www.mozilla.org/MPL (2007). Accessed 22 June 2008

Nordquist, P., Petersen, A., Todorova, A.: License tracing in free, open, and proprietary software. J. Comput. Sci. Coll. **19**(2), 101–112 (2003)

Opensource org: The Approved Licenses. http://www.opensource.org/licenses/ (2007). Accessed 3 April 2007

OSLC, Open source license checker. http://sourceforge.net/projects/oslc (2007). Accessed 22 November 2007

Paakki, J., Koskinen, J., Salminen, A.: From relational program dependencies to hypertextual access structures. Nord. J. Comput. **4**(1), 3–36 (1997)

Palamida. http://www.palamida.com/ (2009). Accessed 13 January 2009

Paulson, J., Succi, G., Eberlein, A.: An empirical study of open-source and closed-source software products. IEEE Trans. Softw. Eng. **4**(30), 246–256 (2004)

Perens, B.: The Open Source Definition. www.opensource.org/docs/definition.php, Open Source Initiative (2005). Accessed 13 September 2005

Pidgin IM client. http://www.pidgin.im/ (2008). Accessed 11 August 2008

Rosen, L.: Open Source Licensing. Software Freedom and Intellectual Property Law. Prentice Hall, New York (2005)

Samoladas, I., Stamelos, I., Angelis, L., Oikonomou, A.: Open source software development should strive for even greater code maintainability. Commun. ACM **47**(10), 83–87 (2004)

Seacord, R., Plakosh, D., Lewis, G.: Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Addison-Wesley, Reading (2003)

Selby, R.: Enabling reuse-based software development of large-scale systems. IEEE Trans. Softw. Eng. **31**(6), 495–510 (2005)

Subversion. http://subversion.tigris.org/ (2008). Accessed 11 August 2008

Tuunanen, T., Koskinen, J., Kärkkäinen, T.: Retrieving open source licenses. In: Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M., Succi, G. (eds.) Open Source Systems. IFIP International Federation for Information Processing, vol. 203, pp. 35–46. Springer, Berlin (2006)

Yu, L., Schach, S., Chen, K., Offutt, J.: Categorization of common coupling and its application to the maintainability of the Linux kernel. IEEE Trans. Softw. Eng. **30**(10), 694–706 (2004)