Open Source Legality Compliance of Software Architecture

A Licensing Profile Approach

Alexander Lokhman, Antti Luoto, Imed Hammouda, and Tommi Mikkonen
Tampere University of Technology, Department of Pervasive Computing
Tampere, Finland
firstname.lastname@tut.fi

***Abstract* — The architecture of a software system is typically described from multiple viewpoints, such as logical, process, and development views. With the increasing use of open source components, there is a new emerging view that should be taken into account: the legality view. The legality view makes explicit the legality concerns of software architecture such as Intellectual Property Rights (IPR) issues and use/distribution terms of the components. These issues are particularly important, when they impose architecturally significant requirements that may influence the architecture. In this paper, we discuss the compliance of software architecture with respect to the legality aspects of open source licenses, and address the various facets of open source legality compliance. We then propose a Unified Modeling Language (UML) profile-based approach and tool to address the legality concerns of open source at the level of software architecture. The technique has been applied to express and analyze the legality view of an industrial case study.**

Keywords-UML profiles; open source software; licensing; software architecture

I. INTRODUCTION

Software architecture has been standardized as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution [26]. Commonly identified stakeholders of software architecture include testers, product managers, users, designers, marketing personnel, and so forth. Architecturally significant requirements resulting from these perspectives commonly include quality attributes such as testability, scalability, understandability, modularity, flexibility, and so on. Thus, the architecture of a software system is represented by multiple views [11]. These views vary in nature and are complementary to each other. Some views show the organization of the code units (e.g., packages and classes). Others show the runtime view of the system (e.g., processes and threads). A third view is to explain how the system is deployed on physical hardware (the deployment view). Each architecture view defines the types of elements and relations that can be represented in that view, and provides means for reasoning about their properties.

We claim that there is a new emerging view to any software system that should be taken into account increasingly often: the legality view. The goal of the legality view is to make explicit the legality concerns of software architecture – such as Intellectual Property Rights (IPR) issues and use/distribution terms of the individual components – in particular when legal aspects are architecturally significant and should therefore influence the architecture. In this spirit, the legality view should clearly state how the legality constraints of the individual architectural elements are satisfied by the overall architecture.

So far, the legality view has been considered in designs to some extent, for instance in terms of encryption and safety requirements (as part of the non-functional view) or data privacy issues (as part of the data view), just to list a few examples. However, due to the increasing use of Free/Libre/Open Source Software (FLOSS) systems freely available on the Internet (e.g., [20]), where licensing issues differ from the conventional proprietary setting and concern the very core of software design, a more holistic view of legality issues associated with open source components is needed [1, 10, 23].

Within the wide spectrum of legality issues of software architecture, the main focus of this paper is to examine open source licenses as primary source for legality concerns in software solutions that involve both proprietary and open source components. We argue that the terms dictated by open source licenses may constrain the architecture of a software system and even act as an architectural driver during design. For software architects, being aware of the rights and duties of the licenses is crucial in producing an acceptable system from the legality perspective. This is an important yet often overlooked piece of the architecture puzzle, which explicitly communicates the architecture's legality fitness for the purpose of providing all the stakeholders with the confidence that the software system does not suffer from licensing violations and shortcomings.

The contribution of the paper is threefold: First, we review the main factors that shall be taken into consideration when addressing the legality compliance issue of FLOSS intensive systems. Second, we introduce the concept of a licensing profile, which is a Unified Modeling Language (UML) profile [13] used to capture the licensing rules and constraints dictated by FLOSS licenses and expressed in architectural design expressed in UML. Third, we present a generic tool named Open Source Software Licensing (OSSLI) [21] that allows for working with licensing profiles.

The rest of the paper is structured as follows. In Section 2, we give a discussion on the legality tensions that arise in

 ISBN: 978-1-61208-304-9

FLOSS intensive systems and discuss the significance of representing legality concerns in architectural designs. In Section 3, we discuss in detail the concept of licensing profiles. A concrete tool environment for licensing profiles is then presented in Section 4. In Section 5, we introduce a real-life design, where the legality view has been incorporated in development from the very beginning to demonstrate the feasibility of the approach. In Section 6, we discuss our approach related to existing works. Finally, in Section 7, we conclude and point out directions for future work.

II. MANAGING OPEN SOURCE LICENSE IN ARCHITECTURAL DESIGN MODELS

In this section, we review licensing constraints dictated by open source licenses and their significance to architectural design.

A. Legality Tension of FLOSS Intensive Systems

When addressing the legality compliance issue of FLOSS intensive systems, there are a number of factors that must be taken into account. These factors not only stem from the nature and terms of the licenses themselves, but also are related to the way the subject software is implemented, packaged, and deployed.

There are plenty of licenses and license models. A straightforward observation when working with open source licenses is that there are many of them – the Open Source Initiative (OSI) [18] lists about 70 licenses. Popular licenses include the GNU General Public License (GPL), the Lesser GNU General Public License (LGPL), the Apache license, the Massachusetts Institute of Technology license (MIT), and the Berkeley Software Distribution license (BSD). The terms of different licenses vary considerably. To give an example, some licenses such as MIT are classified as permissive, granting very broad rights to licensees and allowing almost unlimited use of the licensed code. Other licenses such as GPL are classified as strong copyleft, requiring that works based on the licensed code be published and relicensed to others on the same terms of the initial license. In the middle are weak copyleft licenses such as LGPL, which is a compromise between permissive licenses and strong copyleft. The LGPL grants flexibility to users when linking to licensed software libraries. However, any modifications to the original library should be contributed back on the same terms of the license. Moreover, some licenses have several versions, and there are subtle changes between different versions. A good example is the case of GPL v2 and GPL v3, which are not fully compatible with each other. In addition, the list is by no means complete, and new licenses can be introduced if so desired. For example, a new license can add some minor differences to an earlier one, thus generating a discrepancy between the licenses, or a completely new license can be introduced.

Licenses can be conflicting [5, 8]. To give an example of possible legal incompatibilities between software components, Table I presents a number of open source licenses and their compatibility properties (across open source components themselves) categorized into three cases: mixing and linking is permissible, only dynamic linking is permissible, and completely incompatible.

As an example, a software component under the terms of GPL cannot be directly linked with another under the terms of the Apache license. In this case, the main reason is that GPL'ed software cannot be mixed with software that is licensed under the terms of a license that imposes stronger or additional terms, in this case the Apache license. The Apache 2.0 license allows users to modify the source code without sharing modifications, but they must sign a compatibility pledge promising not to break interoperability, which fundamentally contradicts GPL terms.

TABLE I. EXAMPLE OPEN SOURCE LICENSES AND THEIR COMPATIBILITY

	PHP	Apache	IPL	SSPL	Artistic
GPL	3	3	3	1	3
LGPL	2	2	2	1	2
BSD	1	1	1	1	1

1- Mixing and linking permissible
2- Only dynamic linking is permissible
3- Completely incompatible

Is it derived or combined work? When integrating third party open source components, possibly together with own work, the restrictions and obligations, which the used licenses impose, may depend on whether the work is considered as derived (derivative) or combined (collective) [6]. A simple example of derived work is a modified version of the original software. However, the distinction between derived and combined works becomes trickier when producing new work by combining or linking multiple software components, possibly distributed under the terms of different licenses. Take the example of a software system S, which is the result of linking together an open source component C1 and an own developed component C2. A common interpretation is that system S is considered to be derived work if C1 and C2 link statically (linked during compile or build time) and that S is considered to be combined work if C1 and C2 link dynamically (the two libraries are loaded into a client program at runtime). In a typical case, however, the judge in a court of law makes the final decision. As a matter of fact, the court decision might depend on the specific legal framework of the jurisdiction, in which the case arises, resulting in even more complex legality issues for software developers.

There are thousands of open source components with different risk levels depending on their usage scenario. The number of open source components has grown at an exponential rate during the last decade. This has given software developers a jump on creating software based on existing code. However, many companies are reluctant to use open source software due to the legal risks associated with the use of those components. There have been attempts to classify open source components according to their risk level [7, 28]. Table II gives an example categorization. Four usage scenarios are identified: using the component as a redistributable product, as part of service offering, as a development tool, and for internal use. Three levels of risks have been proposed, as described in the following.

 ISBN: 978-1-61208-304-9

According to von Willebrand and Partanen, [28], valid means that the package can be used as instructed and that no risk has been identified. Possible risk means an interpretation question has been found. This type of issues can be solved by either 1) removing/replacing the problematic files or 2) acquiring additional permissions from the respective right holder or 3) not using the package at all or 4) based on the particular company's risk preferences in such project, a company could accept the risk. Legally, an interpretation question means that an eventual realizing risk would be civil law risk, e.g., monetary (not criminal). Clear risk means that a risk that cannot be interpreted in a way that would not include the risk has been found. This type of issues can be solved only by 1) removing/replacing the problematic files or 2) acquiring additional permissions from the respective right holder or 3) not using the package at all. A company normally cannot accept this type of risk, since it means the possibility of not only civil law risks, but criminal risks. As an example, component Agent++ can be used internally with no risk, has a possible risk when used as a development tool, but exhibits a clear risk when used as part of service offering or a redistributable product.

TABLE II. EXAMPLE SOFTWARE COMPONENTS AND THEIR RISK LEVELS

Comp.	License	Redistribution	Service offering	Development tool	Internal use
Agent++	Agent++ license	3	3	2	1
SwingX	LGPL	3	3	3	3
Libxml2	MIT	1	1	1	1
Cglib	Apache	2	1	1	1

(1) Valid (2) Possible risk (3) Clear risk

Open Source legality interpretations are subject to the way software is implemented, packaged, and deployed [8, 16]. The legality requirements imposed by FLOSS licenses, such as the requirement to publish source code (i.e. the copyleft rule of GPL), may depend for instance on the interaction type of the components (data-driven versus control-driven communication). In the case of mere data exchange between components, there is no copyleft obligation as the two components are considered as separate programs. Also, the copyleft obligation of GPL does not hold if the FLOSS component (or a modified version of it) is deployed as a hosted service. However, if the hosted code is licensed under the terms of AGPL (Affero General Public License) [29], the copyleft requirement does hold, but only in the case of user interaction with the hosted service (in contrast to service to service interaction). In addition, the copyleft requirement of GPL may not hold in case of interactions through standardized interfaces such as the use of operating system public Application Programming Interface (API), in contrast to system hacks that make the two communication components strongly coupled. Finally, compatibility concerns among different licenses may be circumvented if the packaging of components is done by the user instead of building the entire system at the vendor site.

B. *Significance of Legality Concerns in Architectural Design*

This work advocates for the usefulness of representing open source legality concerns in architectural design. This would allow addressing the licensing issues early in the development process. Accordingly, we foresee the following benefits of the approach:

- Raising the awareness of licensing issues for software architects. This could be achieved by offering a communication medium for software architects with respect to legality matters.
- Using architectural models as an early simulation medium with respect to license integrity and validity, which allows the possibility to detect possible violations.
- Aligning and keeping source code and architectural design in sync from the viewpoint of software licenses. This prevents architectural erosion with respect to licensing decisions.
- Legality constraints can be exploited in a forward engineering scenario, for instance to suggest possible architectural solutions to overcome detected license violations. In addition, the constraints can be used to provide guidelines for component selection with respect to possible licenses that can be used.
- Allowing the ability to organize architectural design into license independent models and license specific models to better analyze the effect of licensing decisions.
- Providing a better way of visualizing license violations and their context. It is beneficial to view the violations in graphical models rather than textual source code.
- Studying how the terms of software licenses can influence quality attributes like scalability (e.g., number of users), which are often considered at the architectural level.

According to these points, we propose a visual modeling based approach that enables analyzing license related problems in early development phases while reusing existing models. The approach is designed to work with and support architectural design made in UML.

III. A PROFILE BASED APPROACH

In this section, we present our approach for documenting the legality view of software architecture, assuming that the design model is expressed in UML. Accordingly, we introduce the concept of licensing profiles in detail and illustrate the concept with two example profiles. We start with a brief introduction to UML profiles.

A. *UML Profiles*

The generality of UML constrains its applicability for modeling narrow-scaled domains or problem fields. However, UML offers mechanisms for extending the language. With the help of these mechanisms, it is possible to create an extension that adds more expression power to UML on a certain field or environment. In addition,

 ISBN: 978-1-61208-304-9

traditional UML can be hidden on a lower level so that only relevant properties are displayed.

One of the extension mechanisms in UML is the light-weight profile mechanism, which is based on meta-modeling [13]. Profiles are packages that contain stereotypes, tagged values, and constraints. Stereotypes are a special kind of meta-classes while tagged values are meta-attributes of those classes. Meta-class is a type defined by UML specification. Based on these features, it is possible to define a Domain Specific Modeling Language (DSML) for a certain application field. Profiling is a mechanism of UML, and thus the definitions do not necessarily reflect the actual implementation of the problem but provide a way to express issues conveniently. For example, it is difficult to say how a stereotyped class is implemented in real life, but as a modeling tool it is a convenient way to visualize information.

B. Licensing Profiles

A licensing profile is a UML profile used to attach IPR related information to UML models. Licensing profiles introduce concepts related to the properties of open source licenses in the form of stereotypes and meta-attributes. This allows the user to create a UML model that takes into account what licenses each component is associated with. For example, a software package could be annotated with information such as copyright holder, license type, and the risks associated with its use in different usage scenarios.

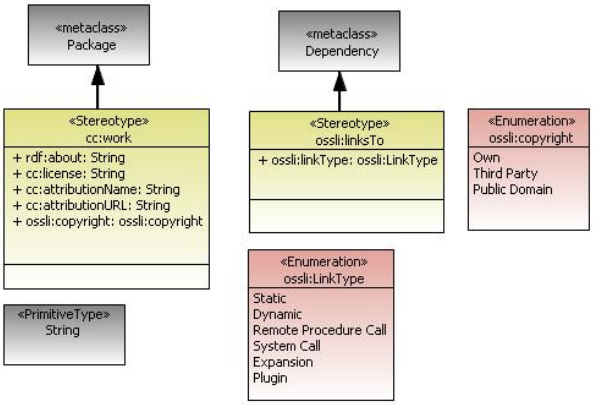

Figure 1. CC REL profile

Figure 1 depicts a licensing profile that is partly based on the specification of the Creative Commons Rights Expression Language (CC REL), a semantic ontology for modeling licenses [2]. The profile makes use of Resource Description Framework (RDF) descriptions for modeling licenses. In addition, the profile introduces concepts, attributes and stereotypes not found in the original CC REL. This is reflected in the naming strategy of the profile – ”cc:” refers to CC REL concepts whereas “ossli” refers to other concepts developed in this work.

For example, the profile defines a stereotype named *cc:work* that corresponds to CC REL class *Work*. The class is defined as “a potentially copyrightable work” in CC REL description. Table III shows the tagged values of *cc:work*. As an example of concepts outside CC REL, the profile defines one stereotype for dependencies. The stereotype is named *ossli:linksTo* and contains one tagged value called *ossli:LinkType*. The tagged value's range is defined in enumeration *ossli:LinkType*. With this tagged value, it is possible to choose a linking type from multiple common types such as static, dynamic, remote procedure call, etc. With the help of CC REL profile, it is possible for example to tell why two open source licenses are conflicting by examining the RDF definition of the license.

TABLE III. TAGGED VALUES OF CC:WORK

Tagged value	Type	Description
rdf:about	String	A standard way in RDF for defining the resource being described. (Uniform Resource Identifier) URI.
cc:license	String	URI to RDF definition of the license.
cc:attributionName	String	The name the creator of a Work would prefer when attributing re-use.
cc:attributionURL	String	The Uniform Resource Locator (URL) the creator of a Work would prefer when attributing re-use.
ossli:copyright	ossli:copyright	Copyright status of the package defined by enumeration ossli:copyright.

A more advanced licensing profile, named OSSLI profile, is depicted in Figure 2. The profile is based on the specification of Software Package Data Exchange (SPDX) [25], recommendations by OSI and other de facto rules for package compliance review [28].

TABLE IV. TAGGED VALUES OF LICENSEDPACKAGE

Tagged value	Type	Description
Copyright	String	Copyright information in free text format.
Description	String	Description of the package in free text format.
License	LicenseType	One or more licenses.
Redistribution	Validity	Validity for redistributing the package.
Development Tool	Validity	Validity for using the package as a development tool.
Service	Validity	Validity for offering functionality as a service.
Internal Use	Validity	Validity for using the package internally.
ID	Integer	Identification for the package.
Ownership	OwnershipType	Ownership of the package.

A fundamental concept in the profile is the stereotype *LicensedPackage*, which extends the standard UML package. *LicensedPackage* has multiple tagged values that are introduced in Table IV. The Tagged values with the type *Validity* are based on package compliance review [28]. Enumeration *Validity* is defined using four values: *Valid*, *Possible Risk*, *Clear Risk* and *Unknown*. The supported licenses are listed in *LicenseType* enumeration, which

includes *Unknown* for packages with unknown license or unexpressed license information. *OwnershipType* is defined in the profile as an enumeration with three values: *Own*, *ThirdParty*, *PublicDomain* and *Unknown*.

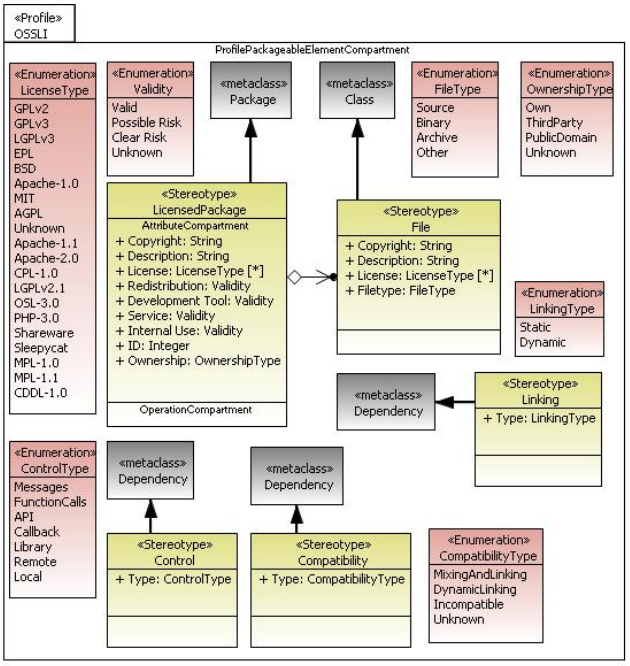

Figure 2. OSSLI profile

The profile shows that *LicensedPackage* is composed of classes that are stereotyped as *File*, which have own tagged values. In addition, the profile defines three dependency stereotypes. *Linking* stereotype consists of one tagged value named *Type*, which tells whether the linking between packages is static or dynamic. Thus, *Type* is defined by enumeration *LinkingType* with values *Static* or *Dynamic*. *Control* stereotype describes control type between packages, such as if the packages communicate with each other using API or remote procedure calls. *Compatibility* is a stereotype designed to mark the compatibility mode of licenses as described previously in Table I.

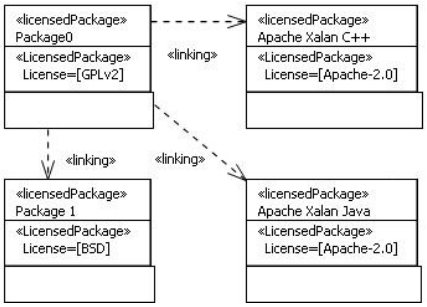

Figure 3. Illustrative example model using OSSLI profile

An illustrative example model using the OSSLI licensing profile is shown in Figure 3. The example consists of four software packages, of which two are owned packages (*Package0*, *Package1*) and two are third party packages (*Apache Xalan C++*, *Apache Xalan Java*). Package0 is linked to all the other packages. The profiled model exhibits licensing information such as license used and linking type information between packages.

IV. OSSLI TOOL ENVIRONMENT

In order to illustrate the use of licensing profiles, a tool named OSSLI [21] has been developed on top of Papyrus modeling environment [22]. The tool is capable of documenting licensing information and managing open source legality concerns in architectural design. In OSSLI, design models are expressed as profiled UML package diagram. Figure 4 depicts the user interface of OSSLI showing the example design model introduced in Figure 3 (middle part of the figure). The left part of Figure 4 shows the selection of the OSSLI licensing profile selected for application.

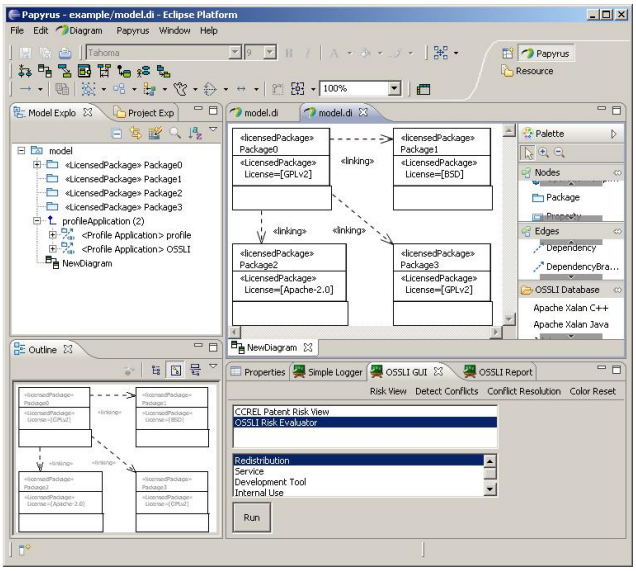

Figure 4. OSSLI user interface

In addition, the bottom part of Figure 4 shows a scenario of running a risk evaluator for product redistribution on the example model. Figure 5 shows the results of the risk evaluation. *Package0* has been reported as risky (marked with red color) while all other packages are without risks (marked with green color). Alternatively, the user could run risk evaluation with respect to service offering, development tool or internal use. The analysis is based on the information of *LicensedPackage*'s tagged values included in the OSSLI profile and introduced in Table II.

 ISBN: 978-1-61208-304-9

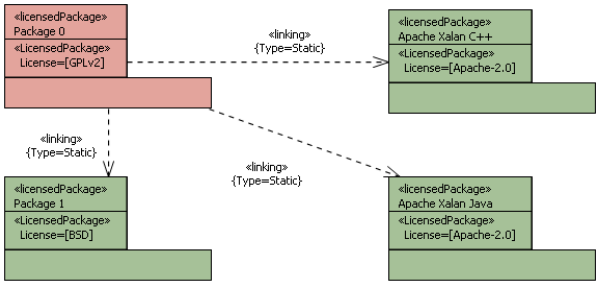

Figure 5. Example risk evaluation of packages

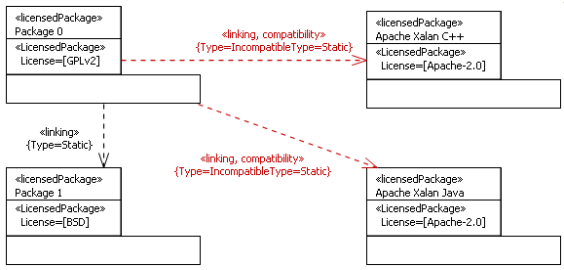

Figure 6. Example of conflict detection

The user could also perform license conflict detection on the profile model. A license conflict occurs when connecting components to each other. This is illustrated in Figure 6. Detected conflicts are presented to the user and are highlighted in the UML diagram in red color. In the figure, *Package0* is reported as conflicting with packages *Apache Xalan C++* and *Apache Xalan Java*. The conflict is reported based on the compatibility values shown in Table 1 and represented using the *Compatibility* stereotype in the OSSLI profile.

V. CASE STUDY: SOLA

The proposed legality view to software architecture has been incorporated in the development of a real-life open source system known as Solutions for Open Land Administration (SOLA) [3]. The project, which is supported by Food and Agriculture Organization (FAO), aims to implement an open source land registration and administration system that will be deployed in at least three developing pilot countries – Nepal, Ghana, and Samoa. The role of the authors of this paper is to provide open source consulting support related to the development of the system, software review, and community building.

The development of the system has been organized in two main phases, a generic phase where the core components of the system are developed by a closed team, and an application phase where the system is adapted to the contexts of the three countries and released to the open source community for further development. A basic project requirement was to reuse the maximum number of existing open source components. This has led to the adoption of tens of open source components with different open source licenses. In addition, a number of other components have been developed by the project team. Figure 7 depicts a fragment of the SOLA system architecture.

As part of the software review task, we have assessed the system architecture from a legality perspective. Example questions we had to address include:

1. Could a GPL'ed icons library be used in the presentation layer?
2. How should the components developed by FAO be licensed? Both individually and as the whole SOLA package?
3. Are there any compliance violations among component interactions?
4. Could the SOLA components be used in proprietary products? If not, how to circumvent this issue?
5. Are there any legality problems related to software compliance with the national e-gov strategies of the pilot countries?

As example answers to the above questions, it was deemed risky to use a GPL'ed icons library as this would trigger the copyleft obligations of GPL, which would be a problem in case the software is used in proprietary systems. Therefore, the library has been discarded.

Figure 8 shows a compliance exercise session for SOLA design model in the OSSLI tool. Analyzing the components interactions and their licenses, several important findings have been observed. First, we identified all possible legality incompatibilities. In Figure 7, a possible risk is mixing LGPL'ed *JasperReports* library and Apache Licensed *Barcode4J*. According to the terms of the licenses developers should use dynamic linking in order to achieve more independence among these components. Other conflict detection risks are highlighted in Figure 8.

As for the components written by the FAO team, we proposed the use of the modified BSD license because it is compatible with all other internally used licenses. Another option we have discussed is to use to use GPL v2. However, the latter option would bring clear risks when combining GPL'ed packages with Apache Licensed libraries (e.g., *Dozer*, *MyBatis*). This is because Apache License and GPL are completely incompatible.

Finally BSD license was also proposed as the main license of the entire SOLA system. This minimizes the legality risks when adopting the software in the pilot countries, and allows commercial companies to develop proprietary software on top of the SOLA system and its components. Furthermore, no conflicts were found between the proposed license scheme and the guidelines of the national strategies of the pilot countries.

VI. RELATED WORK

The fashion FLOSS components are allowed to interact with each other and proprietary software has become an important architectural concern. Present design approaches optimized for the technical aspects of software architecting, such as scalability, reusability, and testability and tend to diminish or even completely overlook the legality dimension

 ISBN: 978-1-61208-304-9

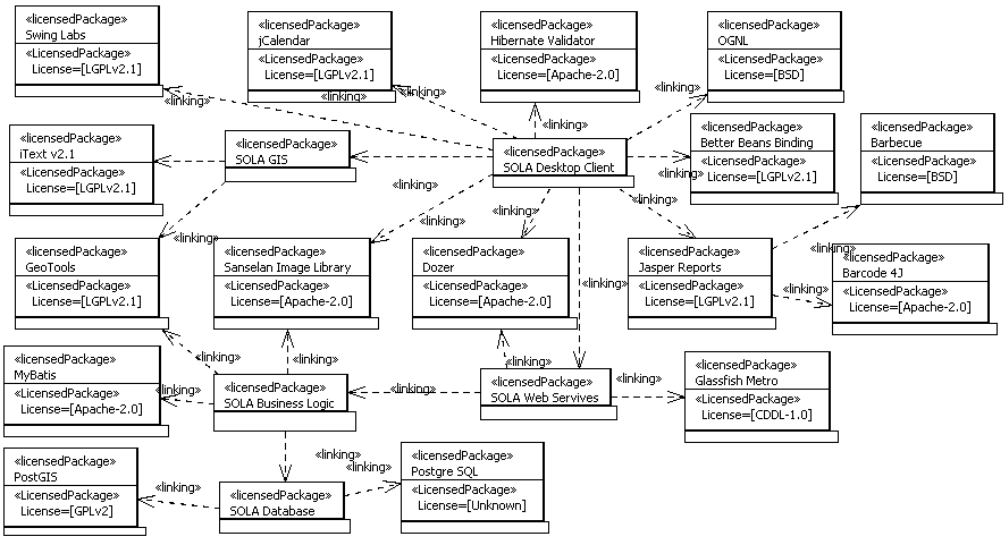

Figure 7. The SOLA Project Legality View

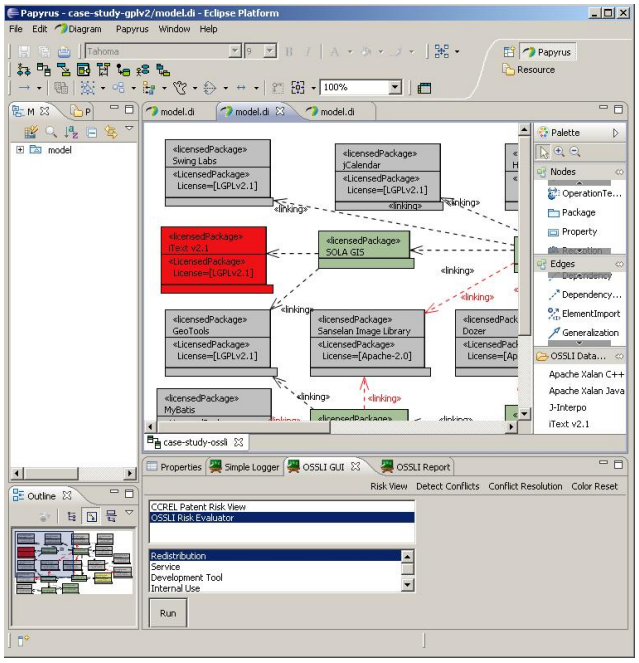

Figure 8. Risk evaluation and conflict detection in SOLA model

that is becoming increasingly important to manage legal dependencies of open source components.

The legality challenge of FLOSS has been partly addressed using so-called license analysis techniques and tools (Table V). Some of the tools provide functionality to identify the licenses through source code analysis. Examples of these tools are Fossology [4], Automated Software License Analysis (ASLA) [27], and Ninka [17]. LChecker [12] provides a similar functionality but takes a slightly different approach. It utilizes Google Code Search service to check if a local file exists in a FLOSS project and if the licenses are compatible. In addition to license identification, Open Source License Checker (OSLC) [19] also provides support for license conflict detection in source code. Dependency Checker Tool (DCT) [14] focuses on detecting compliance problems at static and dynamic linking level on binaries, based on predefined linking and license policies.

TABLE V. A COMPARISON OF OPEN SOURCE LICENSE MANAGEMENT TOOLS

.	Source analysis	License identification	Design analysis	Conflict detection
Ninka	Yes	Yes	No	No
ASLA	Yes	Yes	No	No
Fossology	Yes	Yes	No	No
LChecker	Yes	Yes	No	No
OSLC	Yes	Yes	No	Yes
DCT	No	No	No	Yes
Qualipso	No	No	OWL	Yes
ArchStudio4	No	No	Custom	Yes
OSSLI	No	No	UML	Yes

Compared to the OSSLI tool, the above technique are mostly useful in analyzing ready packaged software systems but give little guidance, with respect to licensing issues, for software developers during the development activity itself. A number of other tools, such as [23] and [1] do provide support for analyzing license conflicts at the architectural level. However, these tools generate own architectural views and have limited integration with the artifacts that software architects work with. The former uses Web Ontology Language (OWL) for modeling open source licenses and the latter uses a custom formal approach. Furthermore, these tools fall short in their ability to support a number of important practices related to license compliance checking. For example, decisions made during the process of fixing the legality compliance problems in the software architecture could also be recorded for future recommendations [15].

 ISBN: 978-1-61208-304-9

There are a number of ontologies and standards proposed for documenting the legal rules and constraints of software systems. Examples include Legal Knowledge Interchange Format (LKIF) [9], Software Package Data Exchange (SPDX) [25], and QualiPSo Intellectual Property Rights Tracking (IPRT) [23]. These works could contribute to the foundation of the proposed legality view, but nevertheless should be enhanced for better ties with the work processes and methods of software architects.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a new perspective to the software architecture, the legality view. The goal of the view is to make explicit the legality concerns of software architecture such as IPR issues and use/distribution terms of the components, which are often important concerns in all software, but need to be further emphasized in open source development due to the different licensing schemes. The view is particularly important in cases where the legality view introduces architecturally significant requirements. In the paper, the benefits of the view were first demonstrated by a small illustrative example and a real-life design, where the different FLOSS related concerns play an important role in the design of architecture.

The consequences of the introduction of a new view to software architecture are many. To begin with, the complexity of legal issues and their effect in software design becomes visible. While making such issues explicit on one hand helps designers to take them into account, on the other hand the design methods and practices must be revised to precisely reflect the new view in an integrated fashion.

In order to express the discussed legality view in a practical fashion, we have proposed the concept of licensing profile, an adaptation of the UML profile concept for the modeling of open source licensing rules and constraints. We then presented tool support for working with licensing profiles.

As future work, we plan to use licensing profiles as a basis for building novel techniques to devise optimal architectural solutions taking into consideration the legality constraints. This could be achieved, for instance, through the use of genetic algorithms [24].

REFERENCES

[1] T. A. Alspaugh, H. U. Asuncion, and W. Scacchi, “Analyzing Software Licenses in Open Architecture Software Systems,” Proc. FLOSS 2009, 2009, pp. 54–57.

[2] Creative Commons. Describing Copyright in RDF. http://creativecommons.org/ns. Last accessed June 2013.

[3] FAO. Solutions for Open Land Administration. http://flossola.org/. Last accessed June 2013.

[4] FOSSology. http://fossology.org/. Last accessed June 2013.

[5] D. M. German, M. Di Penta, and J. Davies, “Understanding and Auditing the Licensing of Open Source Software Distributions,” Proc. ICPC 2010, 2010, pp. 84–93.

[6] D. M. German and A. E. Hassan, “License Integration Patterns: Addressing License Mismatches in Component-based Development, “ Proc. ICSE 2009, May. 2009, pp. 188–198.

[7] F. P. Gomez and K. S. Quiñones, “Legal Issues Concerning Composite Software, “ Proc. ICCBSS 2008, 2008, pp. 204–214.

[8] I. Hammouda, T. Mikkonen, V. Oksanen, and A. Jaaksi, “Open Source Legality Patterns: Architectural Design Decisions Motivated by Legal Concerns”, Proc. AMT 2010, Tampere, Finland, ACM Press, October. 2010, pp. 207–214.

[9] R. Hoekstra, J. Breuker, M. Di Bello, and A. Boer, “The LKIF Core Ontology of Basic Legal Concepts, “ Proc. LOAIT 2007, 2007, pp. 43–63.

[10] International Free and Open Source Software Law Review. http://www.ifosslr.org. Last accessed June 2013.

[11] P. Kruchten, “Architectural Blueprints — The “4+1” View Model of Software Architecture, “ IEEE Software 12 (6), November. 1995, pp. 42–50.

[12] lchecker A License Compliance Checker. http://code.google.com/p/lchecker/. Last accessed June 2013.

[13] F-F. Lidia and A. Vallecillo-Moreno, “An introduction to UML profiles”, UML and Model Engineering, vol. V, no. 2, April. 2004, pp. 6–13.

[14] Linux Foundation. Dependency Checker Tool http://www.linuxfoundation.org/sites/main/files/publications/lf_foss_compliance_dct.pdf. Last accessed June 2013.

[15] A. Lokhman, A. Luoto, S. Abdul-Rahman, and I. Hammouda, “OSSLI: Architecture Level Management of Open Source Software Legality Concerns, “ Proc. OSS 2012, 2012, pp. 356–361.

[16] B. Malcolm, “Software Interactions and the GNU General Public License, “ IFOSS L. Rev, 2(2), 2010, pp. 165–180.

[17] Ninka, a License Identification Tool for Source Code. http://ninka.turingmachine.org/. Last accessed June 2013.

[18] Open Source Initiative. http://www.opensource.org. Last accessed June 2013.

[19] OSLC, Open Source License Checker. http://sourceforge.net/projects/oslc. Last accessed June 2013.

[20] Sourceforge.net. http://sourceforge.net/. Last accessed June 2013.

[21] OSSLI project. http://ossli.cs.tut.fi/. Last accsedd June 2013.

[22] Papyrus. http://www.eclipse.org/modeling/mdt/papyrus/. Last accessed June 2013.

[23] Qualipso project. http://www.qualipso.org/licenses-champion. Last accessed June 2013.

[24] O. Räihä, Hadaytullah, K. Koskimies, and E. Mäkinen, “Synthesizing Architecture from Requirements: A Genetic Approach, ” Relating Software Requirements and Architecture (eds. P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, and I. Mistrik), Chapter 18, Springer, 2011, pp. 307–331.

[25] Software Package Data Exchange (SPDX). http://spdx.org/. Last accessed June 2013.

[26] Systems and Software Engineering – Architecture Description. ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000), 2011, pp. 1–46.

[27] T. Tuunanen, J. Koskinen, and T. Kärkkäinen, “Automated Software License Analysis, “ Automated Software Engineering 16 (3-4), December. 2009, pp. 455–490.

[28] M. von Willebrand and M. P. Partanen, “Package Review as a Part of Free and Open Source Software Compliance, ” IFOSS L. Rev, 2(2), 2010, pp. 39–60.

[29] AGPL, Gnu Affero General Public License. http://www.gnu.org/licenses/agpl-3.0.html. Last accessed September 2013.

 ISBN: 978-1-61208-304-9