TAMPERE UNIVERSITY OF TECHNOLOGY

ANTTI LUOTO

A UML Profile Approach to Managing Open Source Software Licensing

Master of Science Thesis

Examiners: Adjunct Professor Imed Hammouda and Professor Tommi Mikkonen
Examiners and topic approved in the Faculty of Computing and Electrical Engineering Department Council meeting on 11 January 2012

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
LUOTO, ANTTI: UML-profiilipohjainen avoimen lähdekoodin lisenssihallinta
Diplomityö, 50 sivua, 3 liitesivua
Huhtikuu 2013
Pääaine: Ohjelmistotuotanto
Tarkastajat: dosentti Imed Hammouda ja professori Tommi Mikkonen
Avainsanat: Vapaa lähdekoodi, immateriaalioikeudet, vapaan lähdekoodin lisenssien hallinta, UML, Papyrus, avoin arkkitehtuuri

Avoimen lähdekoodin komponenttien hyödyntämisessä komponenttipohjaisessa kehityksessä on useita ongelmia, joihin kuuluvat muun muassa se, miten laillista komponenttien käyttäminen eri käyttötarkoituksiin on ja miten laillista eri lisenssejä omaavia komponentteja on yhdistää toisiinsa. Laittomaksi toiminnan tekee lisenssien ehtojen rikkominen ja riskien havainnointia hankaloittaa esimerkiksi komponenttien kirjava lisensointi ja lisenssien suuri määrä, eikä ohjelmistokehittäjillä ole välttämättä hyvää tuntemusta aihealueesta. Näihin ongelmiin pystytään osittain vastaamaan arkkitehtuurisuunnitteluvaiheessa, mutta lähestymistapaa tukevia käytännöllisiä avoimen lähdekoodin työkaluja ei ole saatavilla.

Tämä diplomityö kertoo avoimen lähdekoodin lisenssiongelmien hallinnasta arkkitehtuuritasolla. Valittu lähestymistapa on UML-pohjainen. Tutkimuksen aikana tehdyn avoimen lähdekoodin lisenssienhallintaohjelmistojen vertailun perusteella UML-pohjainen menetelmä on uudenlainen lähestymistapa.

Tutkimuksen tuloksena syntyi uusi työkalu. OSSLI-työkalu (Advanced Tools and Practices for Managing Open Source Software Licenses) toimii Eclipse-ympäristössä yhteistyössä Papyrus UML-laajennuksen kanssa. Työkalu hyödyntää UML-profiileja ja niitä käsitteleviä plugineja, joiden avulla luodaan avoimen lähdekoodin lisenssien hallintaa tukeva ympäristö. Tässä ympäristössä UML-malleihin voidaan liittää immateriaalioikeuksiin liittyvää tietoa, jota on mahdollista esimerkiksi analysoida automaattisesti eri käyttötarkoituksiin sopivilla plugineilla. Kahden tapaustutkimuksen tuloksien perusteella voidaan sanoa, että UML-pohjaisuus tarjoaa käyttökelpoisen ratkaisun esitettyihin ongelmiin.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
LUOTO, ANTTI: A UML Profile Approach to Managing Open Source Software Licensing
Master of Science Thesis, 50 pages, 3 Appendix pages
April 2013
Major: Software engineering
Examiners: Adjunct Professor Imed Hammouda and Professor Tommi Mikkonen
Keywords: OSS, FLOSS, IPR, Open Source License Management, UML, Papyrus, Open Architecture

There are multiple issues in utilizing third party open source components in component-based development. These issues include, for example, the legality of using open source components in different domains, or the legality of combining various components with different licenses. Infringing the the terms of a license is considered illegal. What makes this a problem, is that there are plenty of licenses, components are licensed in variety of manners and software developers don't have very well knowledge on the topic. While the mentioned issues can be partly detected or solved during the architectural design phase, there are not convenient open source tools concentrating on the issues.

This thesis is about significance of open source license management on architectural level. The chosen approach is UML based which is a novel method for license management according to an open source tool comparison made during the study.

As an outcome of the study, a new tool has been developed. OSSLI (Advanced Tools and Practices for Managing Open Source Software Licenses) tool is built on top of Eclipse based UML platform called Papyrus. The tool uses UML profiles and supporting plugins to create open source license management framework. Two case studies were conducted during the study and results suggest that the tool is feasible in the discussed problem field.

PREFACE

This thesis has been written by Antti Luoto while working at Tampere University of Technology on Department of Software Systems during the spring 2012. I would like to thank TUT (Tampere University of Technology) Open Source Research Group, Adjunct Professor Imed Hammouda, colleague Salum Abdul-Rahman, researcher Alexander Lokhman for ideas, advices and guidance. Thanks to Marko Leppänen for help with proofreading. The work with HHPartners and Validos has also been an eye-opening experience. Thank you everyone for flexible and pleasant working environment.

CONTENTS

ABBREVIATIONS

BSD	Berkeley Software Distribution
CC REL	Creative Commons Rights Expression Language
CDDL	Common Development and Distribution License
CPL	Common Public License
DSL	Domain Specific Language
FW	Framework
GPL	GNU General Public License
GPLv2	GNU General Public License version 2
GUI	Graphical User Interface
HUT	Helsinki University of Technology
HOT	Henkilöstön Osaamis- ja Tavoitetyökalu
IDE	Integrated Development Environment
IPL	IBM Public License
IPR	Intellectual Property Rights
LGPLv2.1	GNU Lesser General Public License version 2.1
LKIF	Legal Knowledge Interchange Format
MIT	Massachusetts Institute of Technology
OSSLI	Advanced Tools and Practices for Managing Open Source Software Licenses
OMG	Object Management Group
OSD	Open Source Definition
OSI	Open Source Initiative
OSLC	Open Source License Checker
OWL	Web Ontology Language
TUT	Tampere University of Technology
RDF	Resource Description Framework
SOLA	Solution for Open Land Administration
SPDX	Software Package Data Exchange
SSPL	StarPound Simple Public License
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XSL	Extensible Stylesheet Language

1. INTRODUCTION

In the field of software engineering, interest in using third party open source components has increased. However, there are multiple legal issues related to developing and publishing programs that utilize third party components with various open source licenses. A license determines the terms how the software is allowed to be used and infringing these terms is considered illegal. These legal issues contain technical issues such as linking between components or issues concerning on the intended use of the software, for example redistributing or offering it as a service. The main problem when working with multiple open source licenses is that they are not necessarily compatible.

In addition to wide range of problems, software developers are not well aware of the problems and juridical assistance and consultation often might be needed. Because of ignorance, missing ability to identify risky situations and lack of suitable tools, the problems are detected too late and software with possible immaterial property right violations might be published, marketed or utilized. It is known that detecting problems in early development phases saves time and money so it would be useful to detect possible IPR (Intellectual Property Rights) related risks and conflicts as early as possible when developing software that utilizes third party open source components. From time to time, this would also help avoiding legal actions from third party copyright holders who think their rights have been violated. A supportive tool would at minimum cut down the manual work for license inspection and let the developers to concentrate on actual development.

The architectural level modeling of the software is usually done in an early development phase, so it would be natural to try to detect the discussed legal risks and conflicts while creating package level UML (Unified Modeling Language) models. Software engineering field is lacking open source tool environments that could support legal risk and conflict detection while working with modeling languages. There are multiple UML tools but the supporting functionality for legal license management is missing in these tools.

The objective of this thesis is to represent the research behind one solution to the discussed problem and to argue the significance of the described approach. OSSLI (Advanced Tools and Practices for Managing Open Source) tool is developed as an example of a solution based on the approach and to actualize how the method works practically. The tool shows what kind of ideas are behind the discussed approach and it also offers a working environment. The idea is to provide a framework for visualizing and automatically detecting IPR related problems in UML environment on an and customizable platform. A central piece of this method is a UML profile that provides a way to attach IPR information within UML models.

The tool is built on top of Papyrus that is an integrated UML extension for Eclipse. Openness and customizability of the OSSLI framework are achieved via using a well-known open source software that provides plugin support as a base, in addition to utilizing architectural design decisions that enable personal customization for example for each organizations own license policies. There nine different types of plugins that can be integrated to the core functionality and the tool designed to work with basically unlimited plugin configurations.

There are related software concerning license management. For example, similar kind of functionality can be seen on Qualipso software but instead of UML, it uses OWL (Web Ontology Language) based semantic description for modeling the software. When compared to that approach, UML is a more popular approach for modeling software. Therefore using an adequate UML profile and customizing UML software should enable a convenient way to detect IPR related issues while reusing UML models.

Two industrial case studies were conducted during the study. The objective of the case studies was to gather experience of the functionality of the OSSLI tool and to represent how the results of the analysis tools are seen in the case study models. The case studies were performed to show the feasibility of the approach in the discussed problem field. In addition, the method and the tool are evaluated in the light of the case studies. The analysis tool found multiple risks and conflicts from both the cases.

The experience gathered during the case studies makes a feeling that the approach and the tool are feasible in the discussed license management work at least to some extent. There are weaknesses considering the usability of the tool and many useful features are missing but these problems could be fixed in future development. However, the tool might be usable and helpful when used as a supportive tool for designing UML models. Existing license management tools and organizational aspects of license management can be combined with the use of the tool.

This thesis consists of seven chapters. The first chapter is this introduction. The second chapter provides theoretical background via defining general terms and problems related to managing open source licenses in architectural design models while the third chapter provides information and a comparison on existing license compliance tools and methods. The fourth chapter introduces related UML technique and UML based approach to license management. Also, two example profiles are introduced in the fourth chapter. The fifth chapter elaborates the functionality of OSSLI tool environment and the sixth chapter discusses the two conducted case studies. The conclusions are presented in the last chapter.

2. MANAGING OPEN SOURCE LICENSES IN ARCHITECTURAL DESIGN MODELS

This chapter defines the terms and context for the study while providing background information for the developed tool. The chapter explains terms such as software license, open source software, and license compliance. In addition, license modeling and architectural matters related to open source licenses are discussed in it.

2.1 Software Licenses

A software license is a collection of terms, clauses, permissions and prohibitions that are defined by a licensor to make an agreement between other parties. In other words, in a license, the licensor explicitly tells how the other party is allowed to utilize licensed software. Naturally different licenses have different terms and characteristics.

According to Välimäki [25, p. 149] licenses have four types of functions.

1. License acts as a juridical contract between the copyright holder and the user, usually so that user gains restricted rights to use the software while the copyright holder obtains the immaterial rights.

2. It is an economical business model for pricing or marketing.

3. It has a technical function so that it can be used to build software development rules.

4. It publishes related information, such as the names of the developers, descriptions of juridical terms or political views.

As an example of one of the most simple open source licenses, the MIT (Massachusetts Institute of Technology) license is represented in figure 2.1. The license is represented in the text format as listed by Open Source Initiative (OSI) [17].

2.2 Open Source Software

Open source software is software that is licensed with an open source license. Open source licenses are widely recognized as licenses that allow redistribution without

Copyright (c) <year><copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Figure 2.1: *MIT License*

cost with available public source code so that anyone could examine the code, make modifications and even redistribute the derived work as long as it happens under the same terms as the license of the original software. The philosophy is that the license should allow the software to be free, open and without constraining restrictions.

A more accurate definition of open source licenses is the Open Source Definition (OSD) by OSI which is a public benefit corporation that is actively involved in open source community-building, education, and public advocacy to promote awareness and the importance of non-proprietary software [17]. There is a wide range of different open source licenses. OSI lists dozens of approved licenses that fulfill the terms of OSD. The most important terms of the OSD are free redistribution, freely available source code and permitting to create derivative works [25, p. 203]. In addition to licenses approved by OSI, there are multiple licenses that don't correspond with OSD or are very rare but still have much in common with the most popular open source licenses.

There is huge amount of diverse open source licensed software available. The scale goes from fully functional operating systems (Linux) and office software (Open Office) to small library components. There are hundreds of thousands of open source projects registered in SourceForge.net [22] which is the leading resource for open source software development and distribution. [22] It has been estimated that 80 percent of all commercial software contains open source components in 2012 [3].

However, open source is not just about source code. Nowadays it is often seen as a business model. Building a community around the software can be seen as one of the primary goals of development.

2.3 Component-Based Development with Open Source Software

Open source software has been generally accepted to be one of the reuse methods in developing software [3]. The usage of third party open source software components is inviting because you don't need to reinvent the wheel, the diversity of components available, the acquisition of the components is free-of-cost and some components are very commonly used, so that they are relatively well tested and actively developed and improved by the open source community. This kind of development is called Component-based development [7].

German & Hassan [7] define software component as any software product, including any "glue" that might be needed for the integration or adaptation of one or more components. Every component has a copyright holder and a license. Components can be reused and modified.

Regardless of all the pros of open source software, not everything is free of problems. Interpreting licenses, intended usage of software, multiple licenses containing packages and architectural level technical decisions create their own problem field that needs both juridical and technical understanding. Perhaps the most unwanted situation in utilizing third party open source components is to be targeted with legal actions because of using the software in a prohibited way.

Chang et al. [3] list reasons for difficulties and ambiguity on using open source software: lack of inspect process and guidelines on own source code, lack of a management infrastructure to support the process and guidelines, lack of comprehensive knowledge resources for license compliances, lack of developers' active mind-set on open source software and the lack of guarantees on quality of some open source software. With a proper set of license management software, these problems could be at least partially resolved.

There are also common misconceptions in using open source that sometimes make people to believe that open source components are harmful in a way that it not true. For example one common myth is that open source code contaminates the company code as if it was a virus. The idea behind that is wrong understanding of the copyleft. Besides, all open source licenses don't even have copyleft clause and it doesn't affect software that is provided as a service or used only internally in the company. According to another myth, all the modifications need to be published, which is false as well. Any of the known open source license doesn't require responsibility for publishing. However the code needs to be available for the party that obtains the modified software. Also, a common false belief is that open source licensing is confusing and causes risks. Actually, there are licensing in many levels and some are more clear than others. In addition, the licensing issues can be solved with work. If one pays attention on the licenses, it is likely that the risks are minimal. [26].

As it can be seen, better understanding and practices for open source related decisions are needed so that misconceptions won't prevent in vain using open source software. This knowledge is useful to both developers and management.

2.4 License Compliance

The license compatibility comes interesting when components with different licenses, integrated in a way or another, compose a whole that is a new derivative work (product) that is being, for example redistributed or offered as a service. Different licenses have different clauses, permission and prohibitions that can violate each other. German & Hassan [7] call the challenge of combining components with different licenses "the license-mismatch problem".

License compliance problems exist amongst open source licenses themselves and not only between open source and proprietary licenses. According to Rosen [21] "open source code licensed under one approved reciprocal license may not be used in a project licensed under another approved reciprocal open source license." A typical example would be that code licensed under a license with a copyleft clause, for example GPL (GNU General Public License), has been modified and redistributed under BSD (Berkeley Software Distribution) license. BSD licensed software can be re-licensed under a proprietary license which is against GPL license definition. [10].

License compliance problems exist also in situations where proprietary licensed code is included into code under an open source license. As a consequence this puts the open source community at risk and could also cause damage to proprietary software company who owns the copyright. Even legal actions could be considered. [10].

It is not easy for the users of open source components to avoid legal issues and concerns. Multiple major software companies have been targeted with juridical procedures because of their use of open source. It is likely that this kind of activity can be seen in other companies as well. [3, p. 2].

License compliance issues introduce legal risks to both open source society and proprietary software companies. The best option would be to avoid risks, which obviously is not easy and always possible. In practice, we can try to confront and minimize the potential risks and try to manage and cope with them. [10, p. 53]. License management software and other organization level practices can help dealing with the the problematic situations.

2.4.1 Interpreting Licenses

What makes it hard to understand licenses and their behavior, is that the software licenses are written in ‘’legal languages” and are indeed difficult to read and understand. Therefore it is a task for a lawyer rather than for example a software developer. [10]. More reasons for the complex understanding and analysis of the licenses include the number of license types, variants, versions, and various grants in the license texts. Alspaugh & al. [1] state that “licenses are often incomplete or hard to understand” possibly meaning that licenses do not contain all the relevant information to conveniently work with them. Different interpretations of the licenses exist and are under discussion.

Even though license texts were easy to understand, it would be useful to automate the compliance detection operation as typical software consists of multiple different components that can be licensed individually. That could also reduce the need for legal assistance in certain open source license decisions.

2.4.2 Component Configuration

According to Alspaugh et al. [1], the configuration of a system also affects the overall license. For example, it makes a difference whether the components are statically linked during the compilation or dynamically linked at run-time. Also, other architectural and maintenance related decisions, such as alternative component interconnections and component replacement should be noted when determining the overall license of the system.

To give an example of possible legal incompatibilities between software components in relation to component configuration, table 2.1 presents a number of open source licenses and their compatibility properties categorized into three cases: mixing and linking is permissible, only dynamic linking is permissible, and completely incompatible.

Table 2.1: *An example open source of licenses and their compatibility*

	PHP	**Apache 2.0**	**IPL**	**SSPL**	**Artistic**
GPL	3	3	3	1	2
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

For 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 software licensed under GPL 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.

2.4.3 Interconnection Types

German & Hassan [7, p. 190] claim that component can be reused in two different usual manners: white-box or black-box. White-box reuse is described as modifying one or more files of a component and distributing them as a part of the software, whereas black-box reuse means using a component without modifications and not necessarily distributing the file along with software.

White-box reuse is likely to form a derivative work but for black-box reuse, determining whether software is derived or collective work depends on the nature of the use and the interconnection types between the component and rest of the software [7, p. 190]. In other words, it can make a difference in license compliance, how the components are connected to each other and how the software is used. German & Hassan [7, p. 190-191] depict five interconnection types: linking (static or dynamic), forking (system calls), sub-classing (inheritance), Inter-Process Communication (service or server) and making a plugin (extending functionality via plugin-architecture). Linking is the most notable when studying license compliance.

2.4.4 Linking Types

In static linking (white-box), after the compilation of source files into object files has been made, a linker copies the required instructions and data of the linked file into the executable. This happens at build time, as opposed to load time. Build time linking means that all the components are integrated before loading the program to memory while in load time linking some parts of the program are integrated when the program is run. In dynamic linking (black-box), the content of the external components is not copied into the executable. Instead, they are included as references to those external components. These references are then resolved and executed at load or run time. [14, p. 8,11].

2.4.5 Other Communication Types

With the other types of linking it is under varying debate whether using these types create a derivative work. It can be questioned if programs using these methods are subject to licensing terms. Remote Procedure Calls are a type of Inter-Process Communication and are related to client-server system architecture and communication. Remote procedure call is a method with which a client can request services from a server. With an interface, the method hides the inter-computer communication aspects of a call so that the function call seems as a normal function call. Popular technologies for remote procedure calls include CORBA, SOPAS and SunRPC. Operating systems provide an interface that allows software to access the resources of the computer. A System call is the method used by a program to request a service from the operating system. Plugins are not necessarily a form of software interaction but a type of component that uses software interactions to operate with other software programs. Plugins are extensions for the host system and there is no standard definition for plugins' interaction with hosts system. Therefore interpreting a plugin as a derivative work depends on the case. [14].

2.4.6 Risk According to Problematic Package

Even though open source package is redistributed under a license approved by OSI, a detailed inspection might reveal such flaws that using the package in certain scenarios, such as redistribution or offering it as a service, might arise legal questions. Willebrand & Partanen discuss these problems in the light of package review process.

To mention a few appearing problems, occasionally identifying the main license of open source package is not as straightforward as one could expect. Related license information or material might emerge as confusing or contradictory. An example of this kind of situation is when there are unclear references to license version: referencing to GPL when there are multiple files with license notice of GPL version 2 and multiple files with the notice of GPL version 3. Redistributed open source packages often contain multiple sub-packages or sub-components. One question arising from these packages is whether the sub-packages or sub-components are compliant in relation to the main license. Additionally, information that relates to patents or eventual export control can cause problematic questions. The introduced problems might lead to situation where an open source component is risky to be utilized for certain usage scenario. However, corrective measures can be used to solve the issue or mitigate risks. [27].

2.5 License Modeling

In general, modeling is often utilized for analyzing or visualization. Modeling can represent the same matter from a different view or abstraction level. As well as other concrete and abstract matters, licenses can be modeled in different ways and methods, for example with meta-models or ontologies.

A license model is a model that expresses the relevant license information in a way that the information is more formal and unambiguous than natural language. This is required for modeling the licenses on computer and it also provides help with automated license management. For example, comparing the properties of a license with the properties of another license can be performed programmatically only if both the licenses are adequately expressed in a license model. [1].

2.5.1 Meta-models

German & Hassan [7, p. 191] suggest that a license is a set of grants: "The conditions for each grant to right r (G_r) can be represented as a set of m conjuncts. All conjuncts should be satisfied for the licensor to receive such grant:

$$G_r(L) = p_1 \wedge \ldots \wedge p_m \tag{2.1}$$

Alspaugh et al. [1, 2] state that they have extended German's model to include semantic connections between obligations and rights. They discuss about a systematic approach that offers a way to analyze license interactions by adapting the licenses to license model. They incorporate their model with an architecture description language called xADL. They also discuss automatic license management in ArchStudio4 which is a software development environment.

In their meta-model "A license consists of one or more rights, each of which entails zero or more *obligations*. Rights and obligations have the same structure, a tuple comprising an actor (the licensor or licensee), a modality, an action, an object of the action, and possibly a license referred to by the action." So the structure for the tuple would be <*actor, modality, action, object, license*>. Possible values for the fields in the tuple, include for example "licensor" or "licensee" for the actor and "may", "must", "must not" for the modality. Action is the verb or verb phrase targeted to the object and license a possible reference to license. An example of a right could be *Licensee - may - distribute - all source code - under BSD.* [1, 2].

2.5.2 Ontologies

LKIF (Legal Knowledge Interchange Format) is a legal core ontology which enables the interchange of knowledge between legal knowledge systems. LKIF is an OWL based technique which complies with Semantic Web. An ontology can help with the process of modeling legal domains. Defining certain terms such as "liability" or "claim" helps with the process of knowledge acquisition. [11, p. 43-44]. As licenses contain legal knowledge, it could be possible to express the license information in LKIF format, and therefore make use of the OWL principles such as automated decision making and interpreting.

Another ontology and semantic web related technique is CC REL (Creative Commons Rights Expression Language) [4] which lets user to describe copyrights and licenses in RDF (Resource Description Framework) format. The schema of the language defines various license related concepts such as “work”, “license”, “permission”, “prohibition” etc.

2.6 Open Source Licensing in Architectural Design

Traditional quality attributes of the software have emerged from the needs of stakeholders such as product managers, testers, users and designers whose interest have been in quality attributes such as testability, scalability understandability and so forth. Hammouda & al. [9, p.1] believe that there is a new emerging view to any software system: the legal view that brings out the legal quality attributes. The increasing use of open source components is the main reason for emphasizing this view. Taking into account that licensing issues consequently affect the architectural design of a system in addition to technical details, these architectural design decisions can be seen as source of finding open source legality patterns.

Hammouda & al. have composed a suggestive list of legality patterns and divided those under three main topics: interaction legality patterns, isolation legality patterns and licensing legality patterns. Interaction legality patterns, which are related to systems that are supposed to be distributed to end users, contain patterns such as using standardized interface calls, linking dynamically instead of statically and using data-driven communication. Isolation legality patterns in turn concentrate on isolating part of the system so that they remain in the use of single authority. These patterns contain such actions as isolating proprietary parts of the program to server and adding a layer between user and open source service. Licensing legality patterns concern on how the different components should be licensed. Examples of these patterns include repackaging so that derived source can be distributed under a new license, using a tier layer that is licensed under a license compatible with the copyleft license while delegating building of the system for end-user. [9].

2.7 Need for License Compliance

As discussed, the concept of license compliance is related to multiple other concepts. In this thesis license compliance is viewed in the light of automated license compliance detection, IPR and legal issues. To represent licenses in computer environment and to detect conflicts, formal addressing of licenses is needed. This addressing is achieved for example by modeling the licenses with meta-models such as German & Hassan [7] suggest. Another method is to utilize ontologies such as LKIF or CC REL.

There are many legal aspects associated with third party component licenses. Technically the problems rise from integrating components with different licenses between each other. For example, it makes a difference whether the components are linked dynamically or statically. There are existing tools that provide different kinds of support for different licensing issues. Some of these tools crawl through the code detecting common license strings and some can detect conflicts between different licenses. License detection can be performed also with organizational management processes that employ legal expertise in co-operation with software developers. In addition to mentioned aspects, more information on license compliance could be built for example by performing studies, gathering experiences, developing tools, creating communities and training developers.

Open source tools that help detecting IPR related issues while designing software are not very common. This thesis proposes a UML based approach that supports detecting IPR related issues early in the development process when modeling software using third party components.

3. EXISTING LICENSE COMPLIANCE APPROACHES

The methods for controlling the license infringements are basically technical methods in addition juridical authorities. Different methods are used depending on what is supposed to be controlled. Instructions and technical inspection methods can be constructed organizationally to minimize the possibility for the infringement of copyrights. [25, p. 229].

A usual case with an open source copyright infringement is that the copyrights are infringed unintentionally. According to Välimäki [25, p. 230-232] there are two different typical situations for unintentional infringement. Either the source code has been copied in contrary to copyright law or the user has not complied with the license terms which both are relevant to development of software with third party open source components. Välimäki claims that these cases are difficult to detect and therefore the infringements are often unintentional. Detecting and predicting these problems beforehand requires that all the source code and the contained licenses are available and identified. For one software component the required work is to analyze the component with text or license detection tools and based on that, to create a juridical review.

It should be noted that that with the help of this method, only restricted knowledge of the possibility of a copyright infringement is obtained. Without source code the juridical review is basically impossible to be executed. [25, p. 232] Tools and methods that help with source code analysis are discussed next.

3.1 Software License Management Tools

In addition to developing a new tool, we introduced ourselves to several open source license management tools. The study for other software was conducted for comparing the features, learning the field and finding ideas. Proprietary software was left out of the study. License management tools are programs that help, in a way or another, managing software licenses. There are different kinds of tools and methods for different kinds of purposes for open source license management. The list of studied software contains OSLC (Open Source License Checker) by Helsinki University of Technology (HUT), Fossology, Ninka, Dependency Checker Tools, Qualipso and LChecker.

Table 3.1: *A comparison of open source license management software*

Software	DCT	Ninka	Fossology	LChecker	HUT OSLC	Qualipso	OSSLI
Source analysis	No	Yes	Yes	Yes	Yes	No	No
License identification	No	Yes	Yes	Yes	Yes	No	No
Design analysis	No	No	No	No	No	OWL	UML
Conflict detection	Yes	No	No	No	Yes	Yes	Yes

Some of the tools concentrate on source code analysis. They read through the software component (package or individual files) and try to recognize licensing related common strings of characters appearing in the source code or accompanying files. Examples of these kinds of open source programs are Fossology, HUT Open Source License Checker and Ninka. These programs are basically used for detecting and reporting the found license information inside a software package. LChecker has similar functionality but slightly different approach. ‘’LChecker utilizes Google Code Search service to check if a local file exists in an OSS project and if the licenses are compatible.” [13] Dependency Checker Tool concentrates on detecting compliance problems at static and dynamic linking level on binaries, based on predefined linking and license policies. The source code scanning for possible matches and confirming the origin of a license is done previously with Source Code and License Identification Tool. [5]

Qualipso software has Semantic Web based approach. They have developed a prototype system to support developers analyzing open source licensing issues. The analysis is based on an ontology of open source licenses using the Web Ontology Language (OWL). The ontology has been used to model multiple open source licenses, to describe software projects and to model relationships between software entities. [20]

A comparison of these tools and their properties can be seen in the table 3.1. The table also contains our OSSLI tool, which helps seeing the field of our study and how OSSLI tool relates to other software. It can be seen, that from other software, Qualipso is closest to OSSLI tool according to license management features.

Furthermore, HUT Open Source License Checker version 3.0 was partly integrated to OSSLI tool. One implementation of Conflict Detection plugin utilizes the conflict detection functionality and license database of the Open Source License Checker for compliance detection. Fossology was used for getting actual practical experience on license compliance review method by Willebrand & Partanen [27]. Other tools were studied more briefly.

There are also proprietary software aiming for similar but more advanced functionality in source code license and compliance analysis. Black Duck Protex, Palamida and ASLA are examples of such commercial tools. Testing these software or comparing our research results to proprietary software is not however included in this study.

The feasibility of Software Package Data Exchange (SPDX) specification for our purposes was also considered. SPDX is "a standard format for communicating the components, licenses and copyrights associated with a software package" [23].

The problem of these open source tools is that they do not necessarily help to detect the compliance problems in architectural design phase which is usually performed early in the development process. Instead, these programs can be helpful when determining the license of certain implemented or released software, which is crucial for the juridical package review process. With Qualipso it is possible to analyze architectural design in OWL but UML is still used commonly in software development without proper license management support.

3.2 License Management Process

Chang et al. [3] introduce a process that can be used to improve open source software usage difficulties. They have identified three phases in which open source software activities should be appended:

1. Design - finding applicable open sources
2. Right after the implementation - inspecting open source software
3. Release - confirming compliances

During the design phase, the licenses of selected open source software should be taken into account, as some of the licenses might have an effect on the later lifespan of the program [3]. Naturally it is adequate to consider the decisions and possible risks as early as possible to prevent future misfortunes. Inspecting open source software and confirming compliances contains investigating the software whether the components are properly used and whether the licenses are compliant [3]. In each of the phases, a proper modeling of the software can help identifying stated problems.

The open source software inspection process by Chang et al. [3] requires two different working roles for the 7-step-workflow: an inspector and a developer. The process contains the following activities:

1. Analyzing source code
2. Confirming license identification
3. Approving licenses
4. Informing inspection result
5. Informing software release
6. Confirming open source compliances
7. Releasing software

Steps 1, 3, 4 and 6 are executed by inspector while steps 2, 5 and 7 are executed by developer. The flow bounces from inspector to developer according to transitions between the steps.

As can be seen, the flow is relatively complex and progresses without concurrency. With a help of pre-inspected software database and automated compliance detection algorithms, some parts of the 7-step-workflow could possibly be bypassed or least lightened. If the developer can identify risky design decisions by himself without the assistance of the inspector during the modeling, it would require less employee resources and time. Therefore it could create the process faster and cheaper. Chang et al. [3] describe a system that takes into account reused code so that it would not require re-inspection and they claim that the inspection efficiency can be improved that way.

3.3 Organizational Aspects

For organizational level support, Chang et al. [3] represent a model with three groups: a software development team, an open source support team and an open source management team. The management team works on the upper level and the support team acts as a link between the development team and the management team. The support team is responsible for the role of the inspector in the inspection process. The working field of the support team also contains defining instructions, strategies, check lists and other knowledge for open source software release. The management team provides overall management view on open source software. If a support team detects an issue on the inspections, the management team provides expert assistance for example on legal interpretation.

Both developers and inspectors can also be trained and educated in open source software issues. For example, topics for training could be general understanding of open source licenses, detailed education on certain licenses, introducing example cases and lessons on using open source licenses. [3].

3.4 Package Compliance Review

Willebrand & Partanen [27] discuss a method called package compliance review that is a part of open source software compliance process that is used for identifying IPR and other related details, in order to report compliant ways to use a certain software component. Their document about review process concentrates on using open source packages in relation to redistribution. In addition to redistribution, other general usage scenarios for open source software include providing a commercial service, using the software as development tool or utilizing it for internal use inside the company.

Package compliance review gives information that can be either generic or related to specific situation. The generic information refers to identifying information such as copyright holder and stated main license whereas specific information results from situations such as linking with other software. [27].

For a package, the review process returns a compliance value that can be one the three possible values: compliant, possibly incompliant and incompliant. Alternative values are also used: valid, possible risk and clear risk. Compliant means that no risks were identified whereas possibly incompliant means an interpretation question has been found. Incompliant indicates that a found risk cannot be interpreted in a way that would not include the risk has been found.

Willebrand & Partanen [27] state that "we have deemed that certain typical situations are considered compliant, if certain defined criteria are fulfilled and contrary indications are not found." In their review part of the process, the goal is to collect all relevant information for the compliant use of the package and analyze the arising legal questions. Source code analysis software is helpful when gathering and reporting the collected information. In addition, project web pages and documentation are analyzed as they often contain useful information.

Table 3.2: *An example software components and their risk levels*

Component	**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

There are three main outcomes of the analysis: the clarity of the main license, the compliance of sub-components and other elements such as patent or export control related information. This gathered detailed information is used to compose a report, along with other helpful data that can be reused. [27]. Table 3.2 presents an example categorization according to discussed method. The table includes software mapped in relation to four identified usage scenarios and three different compliance values.

For wider understanding of the problem field, and to obtain practical experience, the package compliance review method by Willebrand & Partanen was applied for validating multiple open source packages during the study. Fossology (version 1.4.1), also used by Willebrand & Partanen, was utilized during the study to help detecting the license information on software packages. This kind of package compliance review is a method that could be a part of such license management processes as introduced in the previous sections. Our experience suggests that a part of the validation process can be done by software developer with slight understanding of open source licensing. However, legal advice is often needed for verification on problematic situations. An example of such a situation could be when a new license is encountered or the software package's license information doesn't seem consistent.

4. A PROFILE BASED APPROACH

UML (Unified Modeling Language) is a standardized and popular way of modeling object-oriented software. UML standard is defined by OMG (Object Modeling Group). The current version of UML is version 2. UML is a visual language that contains multiple diagram and element types for different abstraction levels and is it applicable for re-use and automatic code generation. Even though UML is usable hand-drawn also, usually it is most efficient when using particular UML drawing software. UML can be extended with profile mechanism.

Out from the software and methods introduced in the previous chapter, we have chosen a UML based approach as it takes the architectural design into account in the early phase of the development. This study discusses two UML profiles developed to support license and other IPR management within UML. Both the profiles use stereotypes to add information to UML package elements. This information is then analyzed automatically with supporting tools in Papyrus environment. Papyrus and analysis tools are discussed later in the thesis.

With a UML profile and a supporting tool environment, it is possible to create a DSL (Domain Specific Language) that is tailored for a certain purpose. This is a way to use UML conveniently for narrow-scaled problem field and to remove not-needed extra features.

4.1 UML Profiles

The generality of UML constrains its applicability for modeling narrow-scaled domains or problem fields. However, UML offers mechanisms for expanding the language. [16]. 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, 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. 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. [16]. Meta-class is a type defined by UML specification. Based on these features, it is possible to define a DSL for certain application field. Profiling is a mechanism of UML and thus the definitions don't necessarily reflect the actual implementation of the problem but provide a way to express issues conveniently. For example it is difficult to say how stereotyped class is implemented in real life but as a modeling tool it is a convenient way to visualize information. A profile must be based on a meta-model, which is in this case UML, and is not very useful standalone [16]. According to UML specification by OMG, profiles should be interchangeable between different tools. It should be noted that profile mechanism doesn't allow changing existing UML meta-model but allows extending it. However, UML offers another extension mechanism that doesn't have such restrictions [16]. This topic is not discussed in this thesis in more detail.

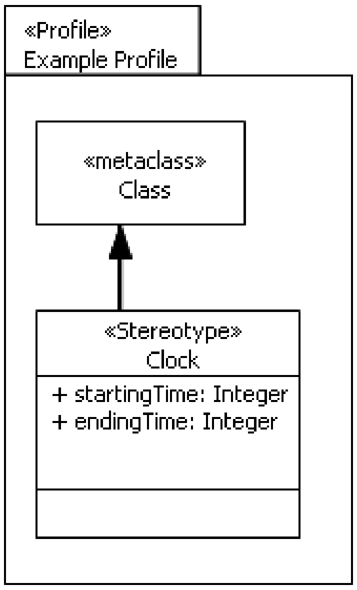


Figure 4.1: *An example of a simple profile*

Profiling in Papyrus is executed via the graphical interface similar to usual UML modeling interface. Papyrus offers a specific Profile Diagram for that purpose. Tool palette for Profile Diagram is customized for profiling purposes. A profile is implemented inside a profile stereotyped package. A clarifying example of a profile and how to use it can be seen in the figures 4.1 and 4.2. In the figure 4.1 there is a simple profile definition with one stereotype extending UML class, whereas in the figure 4.2 there are two classes with the stereotype applied. In Papyrus, to be able to use a profile, it has to be applied to the base model first.

4.1.1 Stereotypes

Stereotype is a class that extends other classes. It tells how the meta-class is extended and how it can be utilized with terminology and notation targeted on a

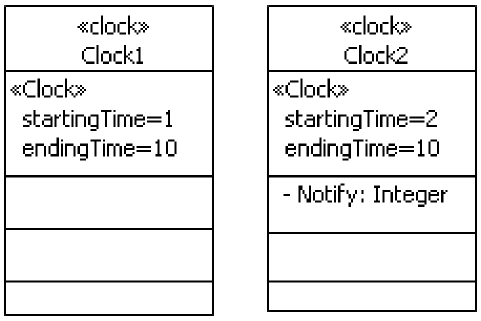


Figure 4.2: *An example of utilizing a profile*

certain domain. As a normal class in UML, a stereotype can contain attributes with values. These attributes are called tagged values. A stereotype definition class uses the same notation as a usual class with the difference that the stereotype is marked with keyword “<<Stereotype>>”. When the stereotype is applied to a model element, the name of the stereotype is shown between “<<” and “>>” notation. [16]. In Papyrus, stereotype that extends meta-class is visualized with Extension relationship. Figures 4.1 and 4.2 show the notation. The black arrow is the Extension relationship.

As an example of a situation where a stereotype can be utilized, we could think of a clock. A clock can be defined as a class according to object modeling and clock’s attributes could be starting time and ending time. It is known, that all the clocks in our example domain have those attributes. Therefore, we would like to define a stereotype “clock” that extends the standard UML class to contain the starting and ending times. After the definition, it is relatively easy to apply the “clock” stereotype to classes wanted with the clock properties. In addition, the classes still have the normal features of UML class, and therefore more methods and attributes can be added. This example is also visualized in the figures 4.1 and 4.2. We can also see that in the figure 4.1 the standard UML class is actually stereotyped with “metaclass” stereotype.

4.1.2 Tagged Values and Constraints

Tagged values are attributes defined by stereotypes, which are added for the elements extended by the stereotype [16]. For example, in the figures 4.1 and 4.2, the starting and ending times of the defined stereotype are called tagged values. The tagged values can also be set to default values if needed.

Occasionally defining constraints is useful for making valid models. The UML definition states that “A constraint is an assertion that indicates a restriction that must be satisfied by a correct design of the system." The constraint specification must evaluate to a Boolean value. One recommended mechanism for defining constraints in UML is Object Constraint Language (OCL). [16]. In the clock example, one could restrict starting time to be bigger or equal than one by stating "startingTime: Integer {startingTime >= 1}". OCL is not used in this study however. Any constraints related to profiles have been expressed in natural language which is an alternative way.

4.2 Example UML profiles

During the study, two example UML profiles were developed to present the idea of attaching IPR related information to UML models and to show how that information can be analyzed in automated methods. One should be able to create profiles with any UML software that has profiling features. OSSLI profile provides a stereotype LicensedPackage that contains for example the name of license as a string. CC REL profile takes a step further and references the licenses with RDF technology. Comparing RDF descriptions of licenses enables a way to tell the reason why licenses are incompatible. One of the reasons for developing two profiles was to show that the tool can work with different profiles and it is not dependent on only one UML profile. The profiles are not meant to be complete but they should provide an idea for the approach. Note that both the profiles are customizable and interchangeable between different tools.

4.2.1 OSSLI profile

OSSLI profile is a UML profile developed for adding license and other IPR related information to a UML model and to support automated analysis of the model making use of the profile. The profile is based on ideas from SPDX [23], OSI [17] and Package Compliance Review [27]. The profile is represented in the figure 4.3. An example imaginary model using the profile can be seen in the figure 4.4. 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.

A fundamental concept in the profile is the stereotype LicensedPackage which extends the standard UML package. LicensedPackage has multiple tagged values that have been described in table 4.1. The table has been represented in more detail in appendix A. Tagged values with the type Validity are based on package compliance review by Willebrand & Partanen and proper values require review process or can be left unknown. Enumeration Validity is defined by 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 license that is not wanted to be expressed. OwnershipType is defined in the profile as an enumeration with three values: Own, ThirdParty, PublicDomain and Unknown. No analysis tool supports Ownership at the moment but it can be used to mark the origins of the package. A one application could be that possibly some analysis is not wanted to be targeted on owned packages.

The profile shows that LicensedPackage is composed of classes that are stereotyped as Files. Files have tagged values as well but at the moment functionality related to files is not available. However, File stereotyped class can be added inside a package to represent notable or problematic files, for example files with noncompliant license.

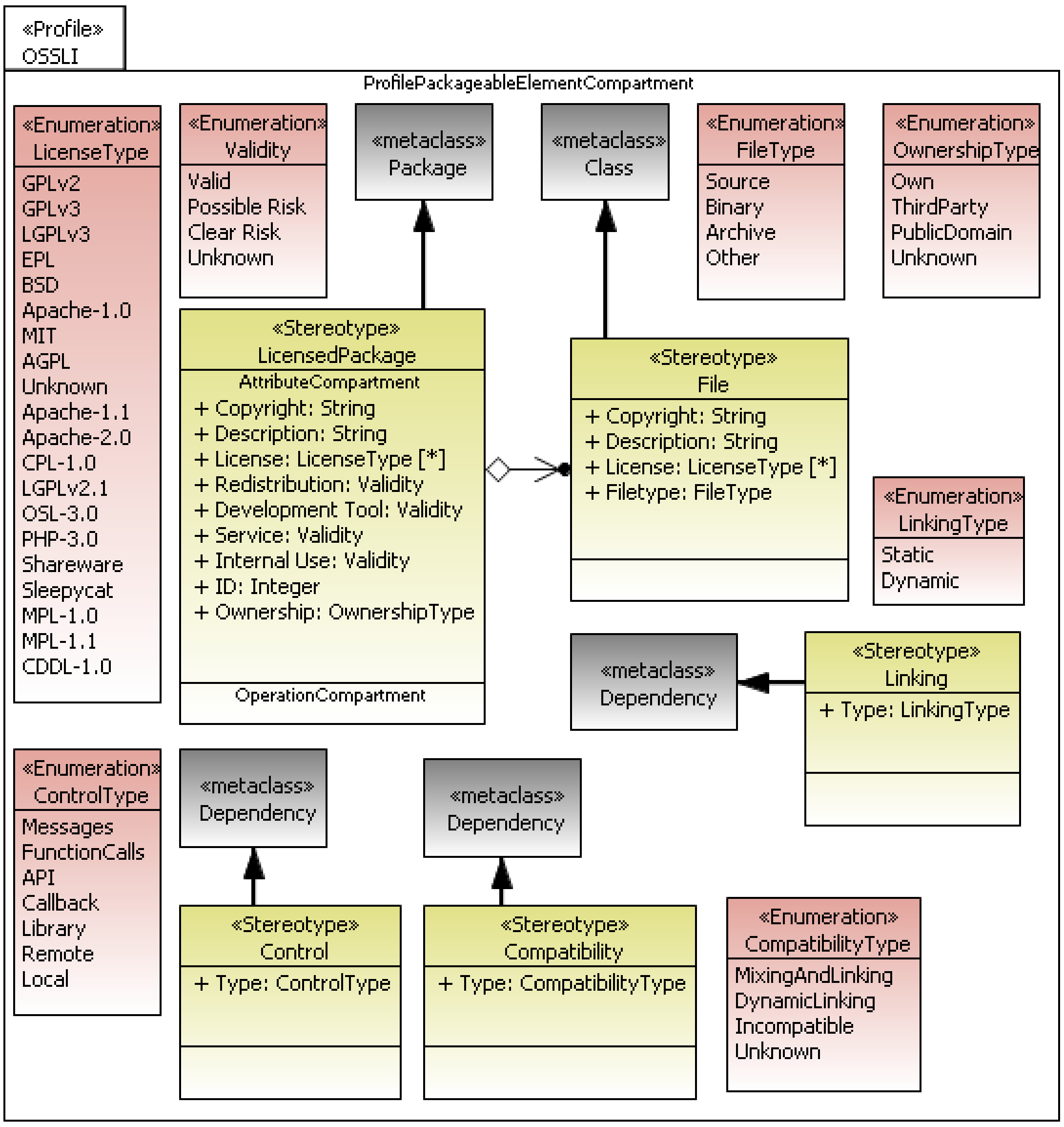


Figure 4.3: *OSSLI profile*

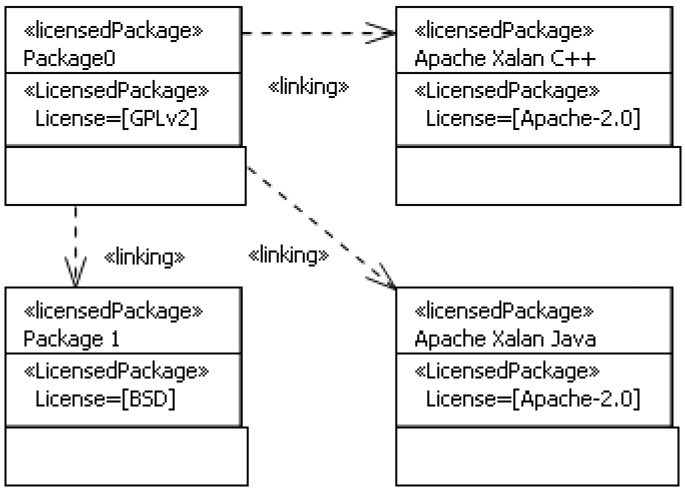


Figure 4.4: *OSSLI profile used in a model*

The profile defines three dependency stereotypes. At the moment, only one of those is supported by the tool functionality. Linking stereotype consists of one tagged value named Type which tells whether the linking between packages is static or dynamic. The type is defined by enumeration LinkingType with values Static or Dynamic. Other dependency stereotypes are not yet supported by analysis tools and therefore not described in detail. Control stereotype describes control type between packages, such as if the packages use each other as API or remote procedure calls. Compatibility is a stereotype designed to support automated tools that can mark whether or not packages can be for example mixed and linked to each other without restrictions.

Table 4.1: *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.

4.2.2 CC REL profile

CC REL profile is another example profile supported by OSSLI tool. As OSSLI profile, CC REL profile is designed to demonstrate the chosen profile based approach. CC REL profile supports the use of CC REL, a semantic ontology for modeling licenses. The profile not only takes concepts from CC REL description but it also defines attributes and stereotypes not found from the original CC REL. The profile makes use of RDF descriptions for modeling the licenses. The naming practice of the profile takes namespaces into account. "cc:" namespace references to CC REL concepts and "ossli" namespace to concepts developed during the study. The profile can be seen in figure 4.5.

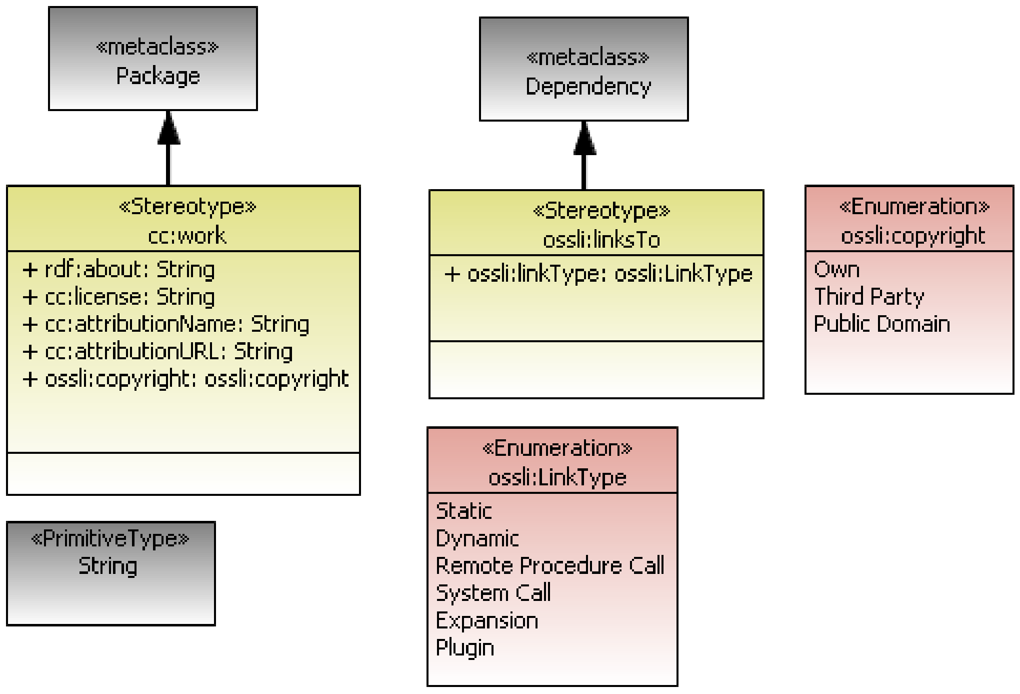


Figure 4.5: *CC REL profile*

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 [4]. Table 4.2 elaborates the tagged values of "cc:work".

As can be seen in the figure 4.5, the profile defines one stereotype for dependencies. The stereotype is named "ossli:linksTo" and it 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.

Table 4.2: *Tagged values of cc:work*

Tagged value	**Type**	**Description**
rdf:about	String	A standard way in RDF for defining the resource being described. 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 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.

The profile might seem more compact and simpler than OSSLI profile but it enables usage of a license model with RDF reference and thus it allows more advanced functionality than OSSLI profile in many situations. For example, with the help of CC REL profile, it is possible to tell why two licenses are conflicting by examining the RDF definition of the license. In other words, it is possible to analyze licenses according to their properties and therefore express issues in more detail. Unfortunately, OSSLI tool lacks support for CC REL profile when compared to OSSLI profile. CC REL profile was not used in the case studies discussed later in the thesis. The profile also requires an available RFD description of a license to be usable.

4.3 Tool Support

To be practically useful, a profile needs to be defined in UML software after which it can be applied to models and the stereotypes can be used. OMG states that profiles should be interchangeable between tools [16]. In addition to actualizing the profile in software, other supportive functionality can be implemented programmatically. With legal matters, customizability can be a relevant need as different legislations and organizational policies exist. Supportive functionality can contain for example automated analysis tools such as conflict or risk detection. Information databases can be also helpful when reusing license data. Learning and decision making functionality might save resources as well. Customization also relates to developing a usable DSL. User interface can be stripped from all the possible extra features to make simple environment for personal or organizational purposes.

5. OSSLI TOOL ENVIRONMENT

OSSLI tool environment has been developed to help with open source license management on UML level. It makes use of UML profiles that enable possibility to attach IPR related information to UML models. In addition to utilizing profiles, it runs on open source platform and it is designed with open and customizable plugin architecture.

The development and running platform of the OSSLI tool is Eclipse Indigo Modeling Tools expanded with Papyrus UML [19] extension that provides a graphical UML working environment for Eclipse. The platform was chosen because it is entirely open source and known to be customizable. In addition, OSSLI tool has been developed on 32 bit Windows XP. Presumably, the software will run on any environment capable of running Eclipse at least with minor customization.

Open source distributed programming Integrated development environment (IDE) Eclipse is known for its extensibility and customizability. A part of the extensibility is based on plugins that associate oneself to Eclipse core via extension points. Eclipse provides a Java API, with which it is possible to create integrated plugins running on Eclipse. Contributing to user interface is also well supported.

Papyrus project including the projects version control provides some documentation that was useful during the research project. Unfortunately, at least some of the documentation is obsolete and has a great variety on quality. Still, most of the Papyrus related problems were solved though not necessarily in very convenient way. One of the most important sources of information was Papyrus Forum hosted by Eclipse. Most of the questions asked by us were answered on the forum and there seem to appear Papyrys related posts daily. Some other parts of the Eclipse Community Forums were useful as well. As Papyrus and Eclipse take advantage of Eclipse Modeling Framework, Graphical Modeling Framework, Standard Widget Toolkit and UML2 styled components, studying them and reading the documentation was part of the process. Generally those components are relatively easy to utilize in the Eclipse environment. Papyrus mailing list is considerable source of information as well.

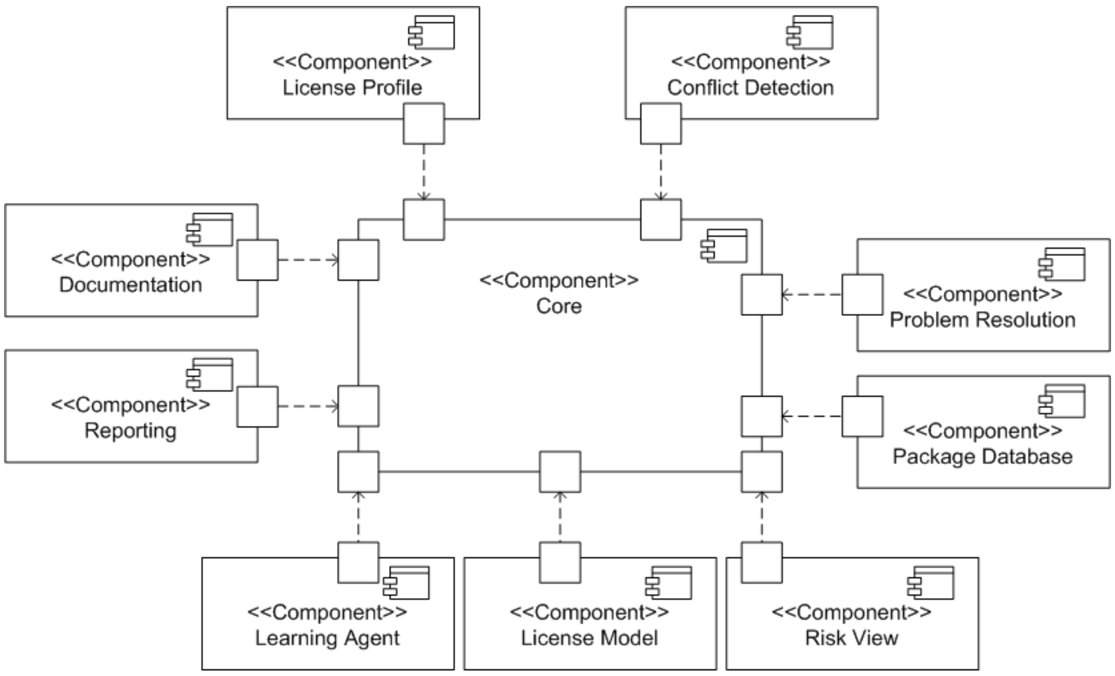


Figure 5.1: *OSSLI tool architecture*

5.1 Architecture

OSSLI tool implementation consists of nine different types of plugins: *Conflict Detection*, *Problem resolution*, *Package Database*, *Risk View*, *License Model*, *Logger*, *Reporting*, *Profile* and *Help*. In addition to these nine plugins, there is a *Core* plugin that binds the other plugins together. Each of these plugins' roots can be seen on the table 5.1 which describes the scientific background behind these components. A part from *Core*, each component is associated with an extension point. The architecture is made extensible so that the tool is able to work with different plugin configurations. The overall architecture of the main components can be seen in the figure 5.1.

5.2 Implementation

A DSL can be created by using the profile mechanism of UML [16]. The profiling in Papyrus is utilized with model based approach that is similar to drawing normal UML diagrams. Because of this, creating a profile is relatively straight-forward for those familiar with Papyrus GUI (Graphical User Interface). Basically creating a profile requires dragging the stereotype elements to diagram, naming them, adding tagged vales and enumerated values, and describing the dependencies and constraints. In other words, it is about defining semantics.

Component	Description	Resource
Core	Handles interactions between the application model, licensing information and the user.	
License Profile	A UML extension to include license information.	[23, 17, 27]
License Model	It describes in computable format the clauses, restrictions, rights and the interdependencies of a license.	[1, 2, 11]
Package Database	A repository containing a list of packages with license, copyright and other IPR related information	[22, 23]
Risk View	Assess legal risks related to use of component for variable purposes re-licensing, sale, internal use etc.	[1, 11, 8, 27]
Conflict Detection	Analysis whether license terms of different licenses conflict when linked or interconnected with another way into the same software.	[1, 24, 6, 18]
Problem Resolution	Suggests operations that can be performed to remove license conflicts from model.	[7, 9, 15]
Learning Agent	Records user actions so that they can be later used to improve program performance.	[9]
Reporting	The analysis results from the different components can be output in different formats.	[6, 24, 18]
Documentation	Provides a way to linking to internal and external documentation on open source licensing concerns.	[12]

Table 5.1: *Scientific background of OSSLI tool components*

To make a profile redistributable and more usable, an own plugin for launching the profile is needed. With the help of this plugin, the profile can be automatically loaded every time Papyrus is started, which allows easy and effective use. Exclusive instructions for defining a profile are located in a document called "Draft Tutorial for Profile usage in Papyrus". When taking the profile in use, it must first be applied to model. After that, the defined stereotypes and other properties can be used on the model elements.

As a proof of concept, two profiles were defined during the OSSLI tool development. The first one is based on concepts by SPDX [23], OSI [17] and Open Source Legality Patterns [9] while the second one is based on CC REL [4] and is designed for supporting OWL techniques. Both of these profiles are described more in detail in chapter 4.

Tool palette is located on the right-hand side in Papyrus GUI. The palette contains the possible addable UML elements, for example classes, packages and associations, that can be drag & dropped to a certain type of UML diagram. OSSLI tools are designed to work at least with class diagrams so the basic tool set for class diagrams is recommended to be visible.

Papyrus provides a special tool for palette customization but it turned out to be

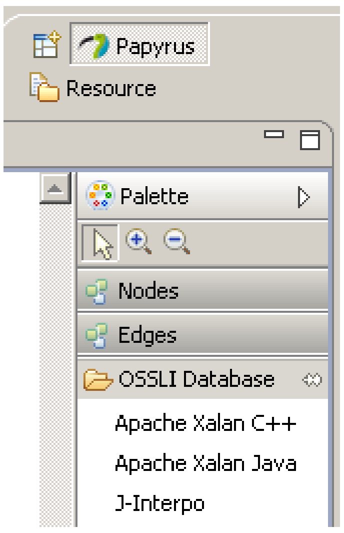


Figure 5.2: *OSSLI database loaded to palette*

insufficient for the needs of loading and visualizing large package databases automatically. The palette also needed to be extended so that the tools could provide a way to add elements with predefined values. In other words the built-in customization tool is only helpful for simple manual palette customization. Therefore, the database plugin needed some programmatic support for the implementation of the features.

As a proof of concept, two *Package Databases* were implemented during the development of the OSSLI tool. Both databases are XML (Extensible Markup Language) databases. The first one is entirely developed by us with similar concepts to OSSLI profile. The second one utilizes SPDX file with multiple package definitions. Both of these database plugins are associated with OSSLI profile.

When "load database" action is executed from the upper OSSLI-menu, the user is prompted with a dialog asking for selecting a component database. After the selection, a *Package Database* is then loaded and it adds a new tool drawer to the default tool palette. The figure 5.2 shows the location of the drawer which contains the packages loaded from XML database file showing their names. These packages can then be drag & dropped to a diagram as normal tool palette elements. The added package will appear as a UML package stereotyped as "LicensedPackage" that contains predefined license information etc. Using these databases is much faster than adding the licensed package information manually by hand every time. This is especially efficient with database that contains often used packages.

License Model plugins are designed for giving a license a form that allows it to be processed by computers. Modeling a license contains work such as dividing the clauses and prohibitions and presenting them in a certain data structure. This

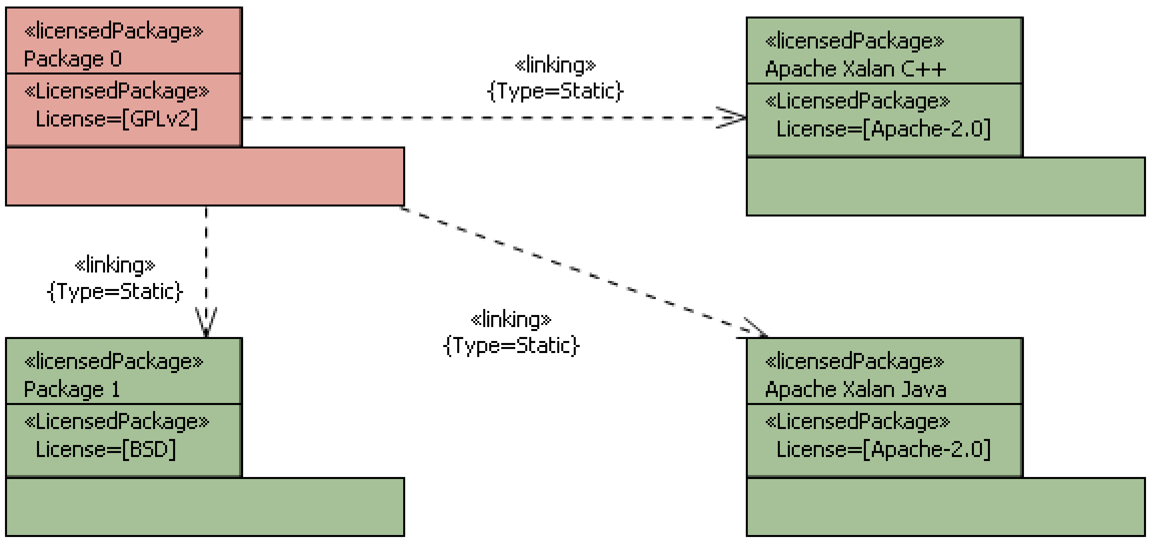


Figure 5.3: *An example of risk view*

enables for example comparing the properties of a license with each other and making different kinds of interpretations: same license can be modeled in multiple ways. License models don't provide any straight user communication features but license models are used by other plugins. Many OSSLI tool operations can be executed without license models as well. In OSSLI tool there is a license model developed with concepts of CC REL that is extended with a few additional clauses. For tools taking advantage of this model, using CC REL profile is required.

Risk View plugins' purpose is to detect risks that are related to single licensed packages whereas *Conflict Detection* plugins are more concentrated on detecting conflicts by taking the dependencies into account. Found risks are shown on overview dialog and indicated with colors on the diagram. In an overview dialog, the user is prompted with an option to create an XML report. A result of risk view analysis on a simple example can be seen in figure 5.3. In the figure, “Package0” has been analyzed as risky while all the others are valid and therefore without risks.

Our team developed two *Risk View* plugins; OSSLI Risk Evaluator and CC REL Patent Risk View. The first one is accompanied with OSSLI profile and needs information of the intended use of the software that is being analyzed. For example user can choose whether to analyze in terms of redistribution, acting as a service, acting as a development tool or being in internal use. The analysis is based on information included with OSSLI profile's LicensedPackage stereotype's tagged values. The results of the analysis are shown on different colors on the model. Red indicates risk, yellow indicates possible risk, green indicates clear of risks and gray indicates unknown. This information is based on manual work by for example package review process.

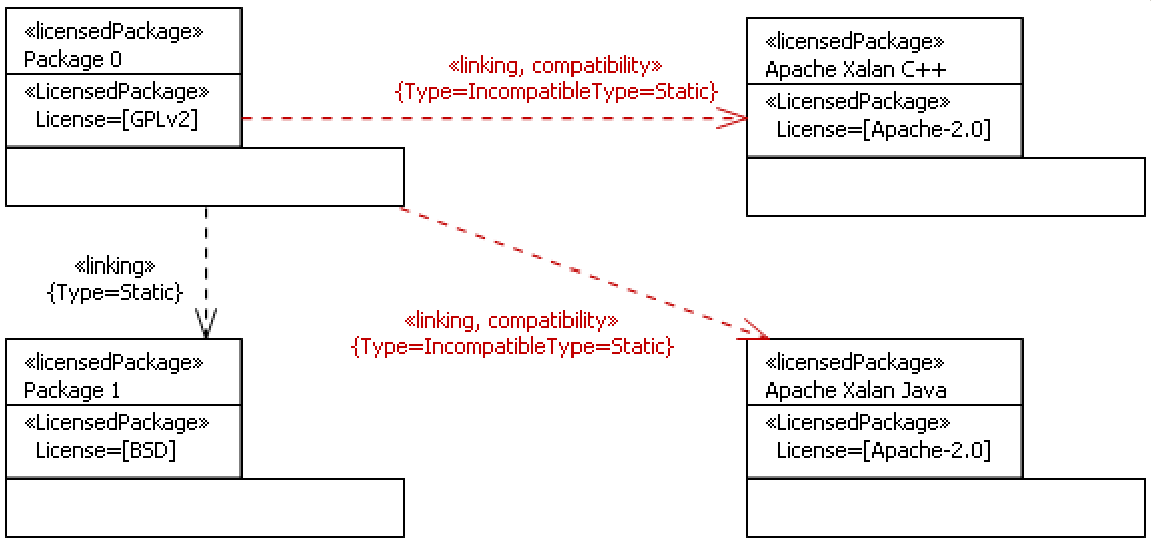


Figure 5.4: *An example of conflict detection*

The latter plugin (CC REL Patent Risk View) is designed for CC REL profile and it analyzes and visualizes whether packages have patenting risks. For example Apache 2.0 license contains patent claim and therefore there is risk trying to patent Apache 2.0 licensed packages. After the analysis, the risky package is presented in red and free-of-risks package is presented in gray. With both the plugins, the analysis can be performed on selected elements only or for whole the model, and they can be executed via upper OSSLI menu, OSSLI GUI or right-click popup menu.

Conflict Detection plugins are designed to recognize license conflicts that occur when connecting components to each other. In other words, these plugins detect components' interconnection related licensing issues. Found conflicts are reported on overview dialog similar to *Risk View* with question about generating a report. Finally detected conflicts are visualized with colors on the diagram as can be seen on figure 5.4 which represents an example of conflict detection analysis results. In the figure "Package0" conflicts with packages "Apache Xalan C++" and "Apache Xalan Java". Compatibility stereotype with value Incompatible has been added to conflicting dependencies along with red color.

Four different *Conflict Detection* plugins were developed during the study: SoberIT, CC REL Detective, CopyleftChecker and OSLC 3.0. Similar to *Risk View* plugins, the conflict analysis can be performed on selected elements only, or for the whole model. Plugins can be also executed via multiple GUI elements similar to *Risk View* plugins.

SoberIT conflict detection is based on license compliance matrix developed by SoberIT [18]. The plugin works with OSSLI profile. Basically it reads all the dependencies on the diagram, checks if a dependency has Linking stereotyped applied and whether the linking type is static or dynamic. Then it reads the license data of supplier and client LicensedPackages which is then compared to compliance matrix.

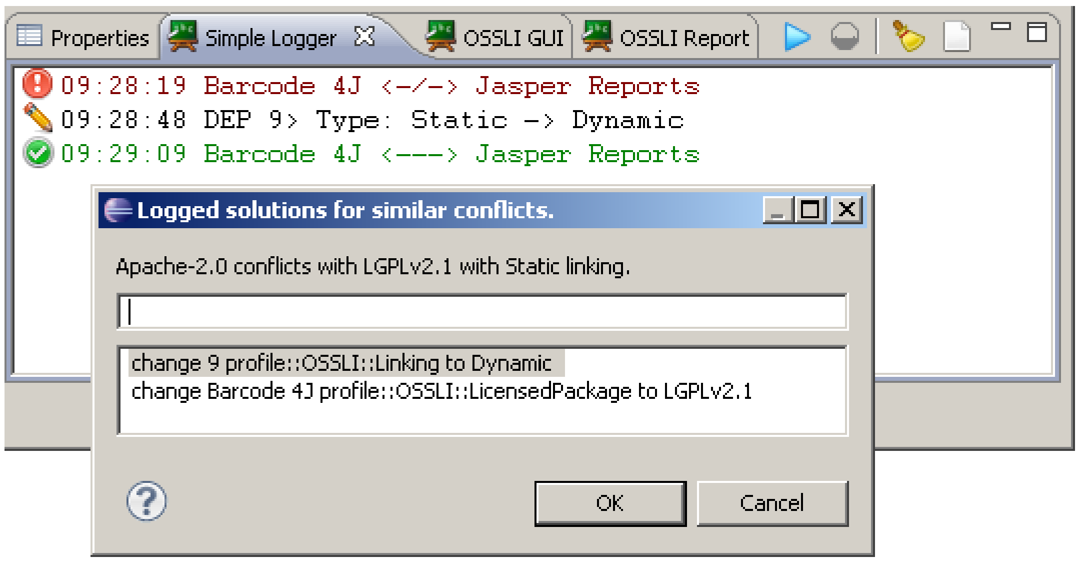
CC REL Detective, which works with CC REL profile, has two running modes; Linking-level and Package-level rule detection. Linking-level analysis detects copyleft compatibility on adjacent linked licensed packages while Package-level analysis detects copyleft compatibility within packages wrapped inside other packages. Conflict color coding in this plugin is different from OSSLI profile based conflict detectors, so that the conflicting packages are marked with red and the dependency related to that conflict is marked with yellow. The reason for this functionality is more clearly seen in Copyleft Checker plugin, as it shows paths that contain multiple packages and dependencies. When compared to SoberIT or OSLC detectors, this plugin has greater power of expression, as it can tell the reason for conflict based on the properties found from the license model. That benefit is obtained via the RDF characteristics of the profile.

Copyleft Checker, also accompanied with CC REL profile, examines the whole tree of linked components taking into account the transitivity of the copyleft clause. This is an enhancement when compared to CC REL Detective that only detects one-step linking conflicts. Therefore it is easier to visualize the conflicting path by coloring the path-end packages red and the dependency path yellow. In other words, red marks the causing packages and yellow marks the path between packages.

OSLC 3.0 (Open Source License Checker) conflict detection plugin is the logic of tool developed in HUT integrated to our system. The integration shows that it is possible to attach third party components to OSSLI tool, which is a proof of expandability. A flaw in OSLC is that it doesn't take dynamic linking into account as the logic is designed for analyzing source code where detecting dynamic linking is not so straightforward.

Conflict Resolver is a plugin that first runs a *Conflict Detection* plugin and then suggests solutions for correcting found conflicts. Finally, the plugin automatically implements the chosen correction. In other words, *Conflict Resolver* in a way continues the flow after conflict detection. It also saves the effort of implementing corrections manually, of course assuming that the solution is reasonable and suitable. An example scenario could be that if a *Conflict Detection* plugin detects that Apache 2.0 licensed package is statically linked to LGPL 2.1 licensed package, one possible solution would be to change the linking type to dynamic. *Conflict Resovelver* suggests this action (among other possible solutions) and if the user confirms, then the linking type is changed to dynamic and conflict disappears from the diagram. This topic is discussed also in relation to *Learning Agent* and the figure 5.5.

OSSLI resarch project has implemented two conflict resolvers. Simple Conflict Resolver is a plugin that reads a log file created by *Learning Agent* and then suggests



```
09:28:19 Barcode 4J <-/-> Jasper Reports
09:28:48 DEP 9> Type: Static -> Dynamic
09:29:09 Barcode 4J <---> Jasper Reports
```

Logged solutions for similar conflicts.

Apache-2.0 conflicts with LGPLv2.1 with Static linking.

```
change 9 profile::OSSLI::Linking to Dynamic
change Barcode 4J profile::OSSLI::LicensedPackage to LGPLv2.1
```

Figure 5.5: *Learning Agent and Conflict Resolver in action*

all the possible solutions for similar conflict situation found from the log. The plugin works with OSSLI profile. CC REL Copyleft Noncompliance Resolution in turn utilizes CC REL profile to suggest solutions for correcting copyleft related conflicts detected by CC REL Detective. These solutions are fixed, so no learning is connected to this plugin. Similar to *Risk View* and *Conflict detection* plugins, the analysis can be performed on selected elements only, or for the whole model. These plugins can be also executed via multiple GUI elements.

Learning Agent supports a *Conflict Resolver* by logging conflict correction actions made by user and thus providing growing database for solutions suggested by *Conflict Resolver*. Example of both of these plugins can be seen on the figure 5.5. In the background of the figure *Learning Agent* has first logged a detected conflict between components “Barcode 4J” and “Jasper Reports”. Then it shows that linking type of dependency named “9” was changed from static to dynamic. After that action, the logger can see that the conflict disappeared. Afterwards in the top window, a *Conflict Resolution* plugin suggests solution for another detected conflict. The first one of the suggestions is the one that was logged by *Learning Agent*. The other suggestion has been learned earlier. Only one *Learning Agent* was developed during the study.

Reporting plugin is a special kind of plugin. It is accompanied with analysis plugins and its purpose is to generate reports of the analysis into files. The only plugin developed during OSSLI study writes XML reports. When the analysis plugin shows the overview dialog, there is an option for generating reports. The idea is to provide a simple XML file that can be for example transformed to more readable format via Extensible Stylesheet Language (XSL).

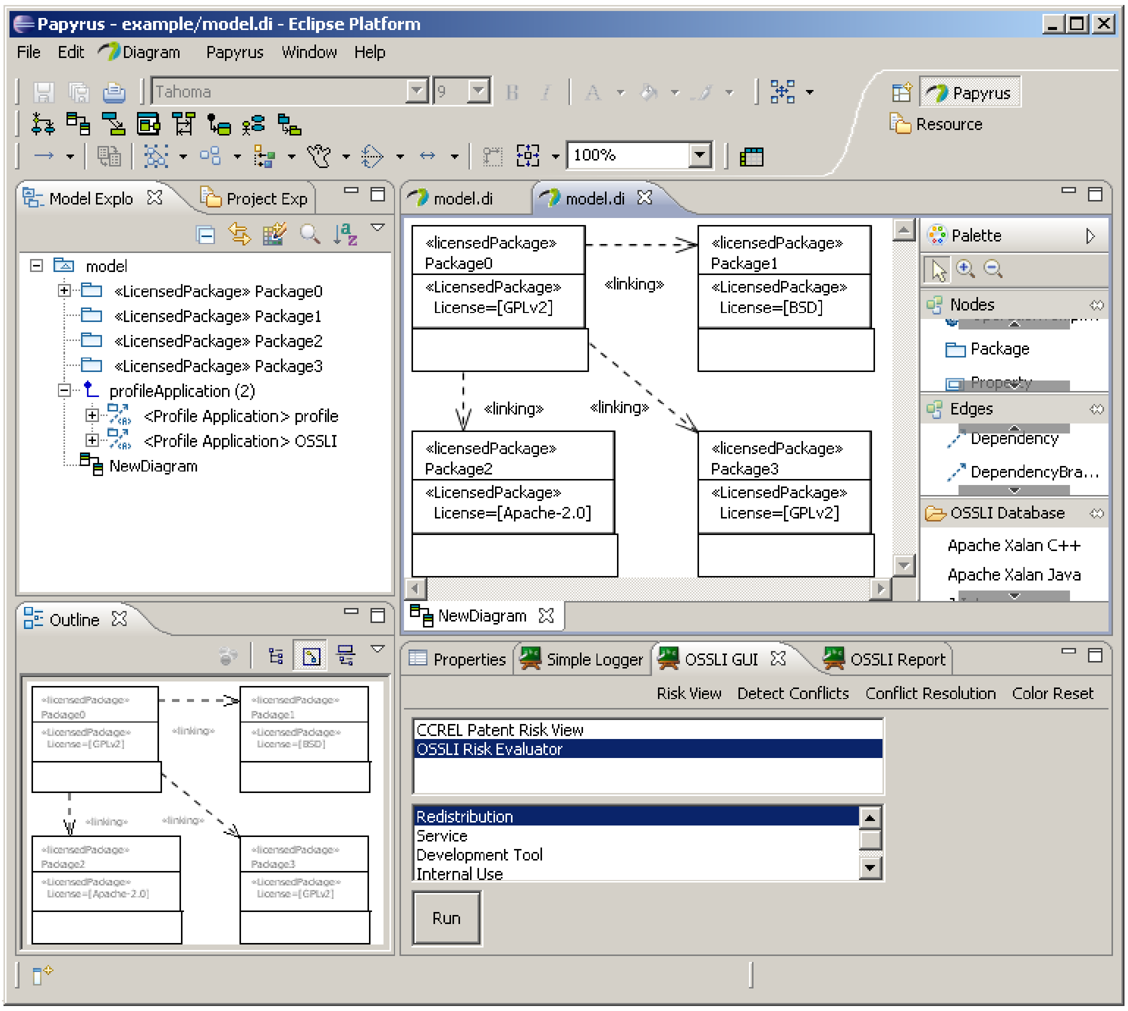

Papyrus - example/model.di - Eclipse Platform
File Edit Diagram Papyrus Window Help
Papyrus
Resource
Model Explo
Project Exp
model
«LicensedPackage» Package0
«LicensedPackage» Package1
«LicensedPackage» Package2
«LicensedPackage» Package3
profileApplication (2)
<Profile Application> profile
<Profile Application> OSSLI
NewDiagram
model.di
«licensedPackage» Package0
«LicensedPackage» License=[GPLv2]
«licensedPackage» Package1
«LicensedPackage» License=[BSD]
«licensedPackage» Package2
«LicensedPackage» License=[Apache-2.0]
«licensedPackage» Package3
«LicensedPackage» License=[GPLv2]
«linking»
Palette
Nodes
Package
Property
Edges
Dependency
DependencyBra...
OSSLI Database
Apache Xalan C++
Apache Xalan Java
Outline
Properties
Simple Logger
OSSLI GUI
OSSLI Report
Risk View
Detect Conflicts
Conflict Resolution
Color Reset
CCREL Patent Risk View
OSSLI Risk Evaluator
Redistribution
Service
Development Tool
Internal Use
Run

Figure 5.6: *User interface*

The purpose of *Documentation* is to provide internal and external links to relevant documentation resources while acting as a help. Documentation is implemented by utilizing the help mechanism provided by Eclipse. Every plugin providing help documentation should contain folder named docs that contains XML file that is used for generating table of contents for the help and subfolder named “html” that contains the actual documentation. Documentation is decentralized so that every plugin contains only documentation of itself. Documentation can be opened from any plugin selection dialog by clicking the question mark on the left-hand lower corner. Under Contents tab, there is node named OSSLI that contains OSSLI tool documentation. Various topics are discussed there.

5.3 User Interface

OSSLI contributes to Papyrus or Eclipse user interface in multiple ways. The figure 5.6 shows a general view on the user interface and its elements. In the figure there is an imaginary example model where OSSLI profile has been applied to, so that IPR related information can be attached on the model.

Contributions to Papyrus GUI by OSSLI include OSSLI menu placed in main tool bar. The menu provides a way of executing *Risk Views*, *Conflict Detectors*, *Conflict Resolution* tools and loading package database. The menu appears on Eclipse's Recourse view. For more convenient user interface, other GUI elements were implemented as well. A separate panel for *Conflict Detection*, *Risk Views*, *Conflict Resolution*, *Logging* and *Reporting* can be opened to the bottom of the screen which can be seen on bottom of the figure 5.6. In the figure, Risk View tab, OSSLI Risk Evaluator and analysis for Redistribution has been selected and run. A general procedure for executing a analysis tool follows these three actions: selecting type of the plugin, selecting the plugin and choosing options and running. However *Conflict Detection* and *Risk View* plugins can be executed via left-clicking the diagram and choosing OSSLI menu actions.

On the right-hand side of the figure 5.6 in the tool palette there is OSSLI Database that lists predefined licensed packages (four packages in the figure) which can be drag & dropped on the model. For visualizing the results of the analysis, some tools change the colors of the diagram elements. Both packages and dependencies can be colored during the analysis to indicate results. After multiple analysis there can be a situation where mixed colors are difficult to understand. That is why there is a button for reseting the colors present.

6. CASE STUDIES

Feasibility of OSSLI tool in the discussed problem field was tested with case studies. The first one was SOLA (Solution for Open Land Administration) case and the second one was HOT (Henkilöstön Osaamis- ja Tavoitetyökalu) case. The purpose of both the case studies was to test the feasibility of the tool in real life software model examples. This target was supposed to be achieved by demonstrating the functionality and analysis results of the tools which were then represented in a form of tables. The demonstrated functionality consists of running risk analysis and conflict detection plugins. In addition to modeling the cases in Papyrus and running analysis tools, the case studies introduce the possibility of performing simulated scenarios of possible alternative license configurations.

In addition to OSSLI profile, instances of Risk View, Conflict Detection, Conflict Resolution, Package Database, Reporting and Logger plugins could be tested with the case study models. For more detailed view on the models, they are found from appendix B.

6.1 SOLA

As a case study, SOLA software, a research project by TUT, was analyzed with OSSLI tool. SOLA software contains 16 third party Open Source components and five components developed by TUT linked statically to each other. Third party components are licensed under various Open Source licenses, such as LGPLv2.1, Apache 2.0, BSD, GPLv2, CDDL 1.0 (Common Development and Distribution License) and PostgreSQL License. Components developed by TUT are licensed under BSD. OSSLI didn't support PostgreSQL License at the time of the study, so it needed to be treated as unknown license. The package diagram of the case study can be seen in the appendix B in the figure B.2. Components developed by TUT don't have license information visible because of readability. All the linking is static.

For worthwhile Risk View analysis for a model using OSSLI profile, some package review needed to be done. The package review database by Validos contained three of the third party components used in SOLA at the time the study was conducted. Validos performs package reviews for open source packages with the method by Willebrand & Partanen [27].

The results of the risk analysis in SOLA case can be seen on table 6.1. In all the usage scenarios, Package iText 2.1 can be seen risky, package Glassfish Metro can be seen possibly risky while package PostgreSQL can be seen valid. Owned packages were treated as valid as they supposedly don't contain third party code and other packages without package review data were treated as unknown. Risk information was manually annotated to the model while the other way would have been to make package database, load it and then drag & drop packages to the model.

Table 6.1: *SOLA risk analysis results*

Component	**License**	**Redistribution**	**Service offering**	**Development tool**	**Internal use**
iText	LGPL	3	3	3	3
Glassfish Metro	CDDL	2	2	2	2
PostgreSQL	Unknown	1	1	1	1

1. Valid
2. Possible risk
3. Clear risk

In SOLA case, Conflict Detection (SoberIT and OSLC 3.0 plugins) analysis found one conflict between JasperReports (LGPLv2.1) and Barcode 4J (Apache 2.0) when the components were statically linked. Results can be seen in table 6.2. The table lists conflicts detected by the analysis by placing the two conflicting packages on a table row. "Client" refers to the package from which the dependency arrow is leaving and "Supplier" refers to the package that the arrow is pointing to. In the table, package and its license are placed in same cell. Note that LGPL and Apache components can be dynamically linked without conflict but in this model they were linked statically.

Table 6.2: *SOLA BSD conflict detection results*

Client - License	**Supplier - License**
Jasper Reports - LGPL	Barcode 4J - Apache

As there was only one conflict and the results are not that interesting, SOLA model was changed so that the components developed by TUT were licensed under GPLv2. With that kind of simulated configuration, multiple conflicts were found. For example, all the dependencies between GPLv2 licensed packages and Apache 2.0 licensed packages resulted as conflicting dependencies. In addition, GPLv2 licensed package was found conflicting with a package licensed under CDDL 1.0. Table 6.3 represents the results.

Table 6.3: *SOLA GPL conflict detection results*

Client - License	**Supplier - License**
SOLA Desktop Client - GPL	Hibernate Validator - Apache
SOLA Desktop Client - GPL	Dozer - Apache
SOLA Desktop Client - GPL	Sanselan Image Library - Apache
SOLA Business Logic - GPL	Sanselan Image Library - Apache
SOLA Business Logic - GPL	MyBatis - Apache
SOLA Web Services - GPL	Dozer - Apache
SOLA Web Services - GPL	Glassfish Metro - CDDL
Jasper Reports - LGPL	Barcode 4J - Apache

According to table 6.3, the simulation with TUT components licensed under GPLv2, the analysis found more conflicts when compared to previous analysis with BSD license. The number of conflicts was eight. Several components were involved with more than one conflict. Examples of these components include SOLA Desktop Client and Dozer. Naturally, the conflict between Jasper Reports and Barcode 4J was detected again as their licenses were not changed.

6.2 HOT

Another case study, that utilizes a model of HOT software, was also conducted. The documentation and source code of the software was obtained from Wapice Oy, a finnish software company. HOT was developed as a course work by students from TUT. The model, constructed by us with the help of the tool documentation, contains 14 packages of which nine are third party and five are developed by students. Third party components are licensed under licenses such as MIT, CPL (Common Public License), Apache 2.0 and LGPLv2.1. The license of the software itself was unknown, so BSD and GPLv2 were used in simulations. A central piece of the software is Spring Framework that wraps most of the other components. This wrapping was treated as statical linking.

The case study model can be seen in appendix B in the figure B.1. Six of the third party components used in HOT case were found from the package review database maintained by Validos. In HOT case, within the same usage scenario as in SOLA case, the results of the performed risk analysis can be seen in table 6.4. In the table Framework is abbreviated as FW. Easymock and Hibernate are risky in all the usage scenarios while all the other packages are valid. Owned packages were treated as valid and other packages without package review data were treated as unknown. As the table 6.5 shows, when owned packages were licensed under BSD, Conflict Detection (SoberIT and OSLC 3.0 plugins) analysis found two conflicts. Conflicting dependencies were found between Spring Framework (Apache 2.0) and Hibernate

(LGPLv2.1) and again between Spring Framework (Apache 2.0) and Junit (CPL). In both the cases, all the linking was considered static. Results are represented in table 6.5. For instructions on reading the table, see the previous case.

Table 6.4: *HOT case study risk analysis results*

Component	**License**	**Redistribution**	**Service offering**	**Development tool**	**Internal use**
Easymock	MIT	3	3	3	3
Hibernate	LGPL	3	3	3	3
Junit	CPL	1	1	1	1
Jquery	MIT	1	1	1	1
Spring FW	Apache	1	1	1	1
Spring FW dispatcher	Apache	1	1	1	1

1. Valid
2. Possible risk
3. Clear risk

Similar to the actions performed in SOLA case, a simulation with owned packages licensed under GPLv2 was conducted. In such configuration all the dependencies between GPLv2 licensed packages and Apache 2.0 licensed packages resulted as conflicting which is equal to results seen also in SOLA case GPL simulation. Results are represented in table 6.6.

Table 6.5: *HOT BSD conflict detection results*

Client - License	**Supplier - License**
Spring FW - Apache	Hibernate - LGPL
Sprint FW - Apache	Junit - CPL

As can be seen, licensing under GPLv2 causes again more conflicts than BSD. The number of conflicts detected by the analysis is 12. Similar to SOLA case, multiple packages are involved in more than one conflicts, Spring Framework being the most conflicting component.

6.3 Use of the Results

Both the discussed cases were industrial cases. The results were used to evaluate the usefulness of the environment. With the results it is possible to estimate whether the functionality is valid. According to manual inspection and available knowledge, the analysis processes produced valid information. At the time of writing this, it is

Table 6.6: *HOT GPL conflict detection results*

Client - License	**Supplier - License**
Spring FW - Apache	Hibernate - LGPL
Sprint FW - Apache	Junit - CPL
Spring FW - Apache	Model - GPL
Spring FW - Apache	DAO - GPL
Spring FW - Apache	Service - GPL
Spring FW - Apache	Controller - GPL
Spring disp. - Apache	JSP - GPL
Spring disp. - Apache	Controller - GPL
DWR - GPL	Service - Apache
JSP - GPL	Sitemesh - Apache
JSP - GPL	DWR - Apache
Service - GPL	Junit - CPL

not known whether the results had any impact on the future usage or development of discussed software, HOT and SOLA. However, next few paragraphs show some possible reactions to the results.

The conflict detection results of SOLA case shown by tables 6.2 and 6.3 suggest that GPLv2 might cause legal problems more probably than BSD as there is more conflicts with GPLv2. Results of risk analysis on table 6.1 indicate that the SOLA contains some packages that could be risky in multiple usage scenarios so caution and preventing actions should be carried out with those packages.

Similar reactions can be adopted to HOT case. The conflict detection results on tables 6.5 and 6.6 show that the model turns out to be more challenging legally when licensing owned packages under GPLv2 instead of BSD. The main conclusions of table 6.4 showing risk analysis results of HOT case suggests that more package review should be conducted on multiple packages which validity was treated as “unknown” on all the usage scenarios. In addition, two packages are clearly risky and in need of caution and supportive actions.

6.4 Evaluation

The objective of the study was to represent the significance of the discussed UML profile based approach. The results achieved via using the environment and the case studies prove that the UML based approach can be utilized for discussed license management work. OSSLI could presumably reduce the burden of developers with automatic IPR related analysis tools and as well reduce the need for legal assistance. Usually UML modeling is done in an early design phase and detecting possible legal risks at the same time would save from troubles in the future. The comparison to other open source license management tools reveals that UML based approach can be regarded as a novel method.

The developed tool was able to detect multiple conflicts and risks from two industrial case study models. Successful simulations with different licenses on case studies argue that the tool can be used as a helpful aid on decision making. License configuration with fewer conflicts might indicate a configuration with less IPR related problems in the future. The supporting features were considered helpful or profitable for future development. Regardless of all the weaknesses of the tool, one could consider the approach convenient and useful in multiple license management occasions. Naturally, the suitability of the case studies can be as well under debate.

Package review is a task that needs some effort and expertise. A proper package review of every third party package in the models requires resources occupied for conducting such a task. Risk analysis of the case studies is therefore partially insufficient, as all the third party packages were not reviewed. In SOLA case only three out of sixteen while in HOT case six out of nine third party components were package reviewed. Still the situation is similar to real world as there is not always enough resources available for everything.

Another problem related to package review is that can it be trusted that the package used in the software is similar enough to the one found from the package review database. In some situations the version number found from the case study model was not exactly the same as the one found from the package review database but still the IPR information gathered from different version was used. This action was performed because the main idea was to demonstrate the functionality of the approach and not necessarily to execute a comprehensive analysis of the model.

As can be seen from the case study models, a clear advantage from the profile based approach is that the license information can be attached to UML elements while re-using package diagram. Naturally this saves time and effort. However, in these case studies the package diagram needed to be generated separately only for this purpose. A more descriptive and efficient case would be one where a package diagram has already been made during the software development and possible found risks and conflicts could be fixed before the implementation. However, the results can be useful as well after the development for example for learning risk free practices.

Another successful matter is that the visualization of the results seems clear and suitable for the intended purpose. While risk view only shows manually annotated information in different form than text, it is useful to combine the information straight to UML model so that it can be visualized among the other license data. In connection with different plugins' visualization, a diagram can contain results of multiple analysis for evaluation.

HOT case study model was constructed primarily according to documentation and secondarily according to source code. Therefore there are assumptions such as everything is statically linked. Even though the model presumably doesn't represent the exact configuration, it is closely related and could represent some software model in theory. There were not similar problems with SOLA case and model was constructed according to very similar model picture. One weakness with both the case study models is that they include only static linking and no dynamic linking. A case study with dynamic linking would have been more comprehensive as dynamic linking is a different situation from the perspective of conflict analysis tools.

As a platform, while Papyrus is a UML tool, it is not the most convenient one for this kind of purpose. Annotating packages and dependencies feels burdensome because of the amount of clicking and usability of Papyrus GUI. In other words, stereotyping packages and dependencies and attaching all the related information is not as straightforward as one would expect. Some usability features were developed for OSSLI but still the platform itself causes multiple problems.

It was noted that using a package database for adding packages to diagram can be a faster method than manually annotating packages with the help of Papyrus GUI. Therefore package database can be seen as useful tool. One weakness of package databases is that they cannot be used vice versa. It would be convenient to be able to add manually annotated packages from the diagram to a package database. That would enhance the reutilization of the modeled packages. A shortcut for adding stereotyped dependencies would also improve using experience. At the moment user needs to add dependency and stereotype individually which is connected to the way UML software often work.

One drawback in the current development level is that not all needed licenses are modeled. For example PostgreSQL license needed to be treated as unknown license and therefore the analysis wasn't as successful partially as one could expect. PostgreSQL license is not one of most common licenses. However, many of the most often used licenses are modeled in the context of OSSLI tool. This problem will be a difficult task to overcome completely as new licenses will be found from time to time. Naturally there are ways to add new licenses to profiles and analysis tools.

It can be under debate if the analysis tools are trustworthy logically or do they work correctly in OSSLI environment. Used conflict detection tools are based on compliance studies made by SoberIT and package review behind risk analysis tool is done by Validos. Therefore trustworthiness of these methods goes beyond our study. According to manual inspection the results seem correct. Utilization and co-operation of other existing methods and theoretical background supports the applicability of the method and tool.

XML reports alone are not necessarily very convenient for humans but it was noted that reporting functionality is working and useful especially in basis of future development. Reports can be re-used for example in connection with XSL transformation to generate a report in a different form. This can be beneficial for organizational redistribution of information.

All the plugins as well as CC REL profile weren't tested during the case studies. CC REL profile could be more useful in some cases, as utilizing it could provide more detailed information on the conflicts but modeling even the most popular licenses for the profile in RDF would have required too much of resources.

It was also studied that Logger plugin can record found and fixed conflicts during the analysis and Conflict Resolution plugin can suggest solutions for similar conflicts according to recorded log file. An example scenario was when the conflict between JasperReports and Barcode 4J was found, the logger was turned on when changing the linking type between the components from static to dynamic. After re-analysis of the original model with a conflict resolution plugin, the plugin suggested the correction and could fix it as well. Still, Logger has somewhat minimal functionality and requires more work to be useful.

7. CONCLUSIONS

The objective of the study was to represent the significance of the discussed UML profile based approach. As a proof of concept, a new open source license management software utilizing the approach was developed. Two cases studies were conducted during the study to demonstrate the features of the tool and to prove the approach to be convenient in the discussed license management work. Other existing license management methods and tools can be used in co-operation with the approach. One example is the package review process by Willebrand & Partanen with which it is possible to provide a database for the tool.

During the study, a new open source license management software called OSSLI was developed. The main idea was to provide a framework utilizing UML profiles to extend certain UML elements with IPR information and to implement supporting plugins for advanced functionality. OSSLI is built on top of Eclipse based UML software Papyrus. The approach of OSSLI is UML based with supporting customizations and contributions for Papyrus environment.

OSSLI tool consists of nine different types of plugins bound together by Core plugin. The tool is built to work with different kinds of plugin configurations so that there can be simultaneously multiple implementations of all types of plugins available. Apart from Core, other plugin types implement features such as risk view and conflict detection analysis. These features and analysis results were demonstrated in the case studies. Executing and development of multiple independent plugins suggests that the tool architecture provides a way to customize the tool for personal needs.

Two different profiles were developed. OSSLI profile utilizes the idea of attaching IPR related information to UML models while CC REL profile that supports RDF is more advanced but still lacks analysis tool support when compared to OSSLI profile. OSSLI profile was used in both the case models, whereas CC REL profile wasn't applicable at the current development phase as it lacked proper tool support and RDF modeled licenses.

The two case studies, SOLA and HOT, both contained multiple third party open source components. OSSLI analysis tools found risks and conflicts from both the case study models. These results claim that OSSLI is a beneficial tool to some extent. OSSLI turned out to be able to detect and visualize IPR related conflicts and risks from both the case study UML models. Successful execution of the simulations with different licenses, gives an impression that the tool could be useful aid with decision making in license management. The suitability for case studies can be under debate for example as the models didn't contain dynamic linking and multiple third party packages were left without package review.

What makes UML based approach and OSSLI different from other software, is that other license management software are not necessarily meant for analyzing software models but rather automatically searching for licenses from implemented software or detecting conflicts from software that already exists. From other software, Qualipso is the most similar to OSSLI but it uses OWL based technique for modeling the software. The approach used by OSSLI, helps detecting IPR related problems early while developing the software model. UML is a popular way of modeling software.

OSSLI still lacks many useful features and much of content. Usability should be also taken into account more for example by reducing the amount of clicking around the GUI. However, the framework can be seen functional and convenient, as the two case studies suggests. OSSLI tool could be seen useful for organizations working with third party open source components in component based development. An open and plugin based structure of OSSLI enables the tool to be customizable for the needs of each organization.

Main ideas for future development include developing a better support for automatic and learning problem resolution by implementing functionality for utilizing open source legality patterns introduced by Hammouda & al., providing wider package review database, implement possibility for adding packages to database directly from GUI, providing a better support for CC REL profile and enhancing GUI usability. Building a community around the software would be an important task in making the software useful and getting more coverage. It is good to bear in mind that caution should be considered always when working with legal matters and automatic decision making.

BIBLIOGRAPHY

[1] Alspaugh, T. A., Asuncion, H. U. & Scacchi, W. Analyzing Software Licenses in Open Architecture Software Systems. FLOSS '09 Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development. IEEE Computer Society Washington, DC, USA, 2009. pp. 54-57.

[2] Alspaugh, T. A., Asuncion, H. U. & Scacchi, W. Intellectual Property Rights Requirements for Heterogeneously-Licensed Systems 2009 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, September 31, 2009. pp. 24-33.

[3] Chang, S., Lee, J., & Yi, W. A Practical Management A Practical Framework for Commercial Software Development with Open Sources. IEEE 7th International Conference on e-Business Engineering (ICEBE), Shanghai, China, November 10-12, 2010. IEEE Computer Society Conference Publishing Services 2010. pp. 164-171.

[4] Describing Copyright in RDF. [WWW]. [Cited 28/5/2012]. Available at: http://creativecommons.org/ns

[5] Linux Foundation. Dependency Checker Tool Overview and Discussion. White Paper. [WWW]. [Cited 28/5/2012]. Available at:
http://www.linuxfoundation.org/sites/main/files/publications/
lf_foss_compliance_dct.pdf

[6] FOSSology. [WWW]. [Cited 28/5/2012]. Available at:
http://www.fossology.org/

[7] German, D., Hassan, A. License Integration Patterns: Addressing License Mismatches in Components-Based Development. IEEE 31st International Conference on Software Engineering, Vancouver, Canada, May 16-24, 2009. pp. 188-198.

[8] Gomez, F. P., Quinoñes, K. S. Legal Issues Concerning Composite Software Seventh International Conference on Composition-Based Software Systems, 2008. ICCBSS 2008, Madrid, Spain, February 25-29, 2008. IEEE Computer Society Conference Publishing Services 2010. pp. 204-214.

[9] Hammouda, I. , Mikkonen, T., Oksanen, V., Jaaksi, A. Open Source Legality Patterns: Architectural Design Decisions Motivated by Legal Concerns In

Proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments (2010), Tampere, Finland, October 6-8, 2010. ACM Press. pp. 207-214.

[10] Helander, N., Aaltonen, T., Mikkonen, T., Oksanen V., Puhakka, M., Seppänen, M., Vadén, R. & Vainio, N. Open Source Management Framework. Tampere 2007, eBRC Research Reports 38. 44 p + Appendixes 48 p.

[11] Hoekstra, R., Breuker, J., Di Bello, M., Boer, A. The LKIF Core Ontology of Basic Legal Concepts. Proceedings of the Workshop on Legal Ontologies and Artificial Intelligence Techniques LOAIT, Stanford, USA, June 4, 2007. pp. 43-64.

[12] International Free and Open Source Software Law Review. [WWW]. [Cited 28/5/2012]. Available at: http://www.ifosslr.org

[13] lchecker A License Compliance Checker. [WWW]. [Cited 28/5/2012]. Available at: http://code.google.com/p/lchecker/

[14] Working Paper on the legal implications of certain forms of Software Interactions (a.k.a linking). 2010. Free Software Foundation Europe, Working Paper. 49 p.

[15] Malcolm, B. Software Interaction and the GNU General Public License. International Free and Open Source Software Law Review, 2(2), pp 165-180.

[16] OMG Unified Modeling Language (OMG UML), Infrastructure. 2009, Object Management Group. 214 p. [cited 2012.1.17]. Available: http://www.omg.org/cgi-bin/doc?formal/2009-02-04.pdf

[17] Open Source Initiative. [WWW]. [Cited 28/5/2012]. Available at: http://www.opensource.org/

[18] Open Source License Checker Wiki. [WWW]. [Cited 28/5/2012]. Available at: https://wiki.ow2.org/oslcv3/

[19] Papyrus. [WWW]. [Cited 28/5/2012]. Available at: http://www.eclipse.org/modeling/mdt/papyrus/

[20] Qualipso. A Protype Decision Support System for OSS License Compatibility Issues. [WWW]. [Cited 28/5/2012]. Available at: http://www.qualipso.org/licenses-champion

[21] Rosen, L. 2004, Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall PTR, Upper Saddle River, NJ, 2004

[22] SourceForge. [WWW]. [Cited 28/5/2012]. Available at: http://sourceforge.net/

[23] Software Package Data Exchange. [WWW]. [Cited 28/5/2012]. Available at: http://spdx.org/

[24] Tuunanen, T., Koskinen, J. & Kärkkäinen, T. Automated software license analysis. Automated Software Engineering 16 (3-4), 455-490, December, 2009. pp. 455-490.

[25] Välimäki, M., P. 2009. Oikeudet tietokoneohjelmistoihin, 2. painos. Helsinki, Talentum. 267 p.

[26] Willebrand, M. 10 myyttiä avoimen lähdekoodin juridiikasta ja riskeistä, Open Solutions 2010, Meripuisto, Espoo 2010. 11 p.

[27] Willebrand, M. & Partanen, M. 2010. Package Review as a Part of Free and Open Source Software Compliance. International Free and Open Source Software Law Review 2, 1, pp. 39-60.

A. APPENDIX: LICENSED PACKAGE

Tagged value	**Type**	**Description**
Copyright	String	Copyright information in free text format. For example, name of the copyright holder and year.
Description	String	Description of the package in free text format. For example additional licensing information.
License	LicenseType	One or more licenses chosen from LicenseType enumeration. There can be multiple licenses for example in cases of dual-licensing.
Redistribution	Validity	Validity for redistributing the package chosen from the enumerated type Validity.
Development Tool	Validity	Validity for using the package as a development tool chosen from the enumerated type Validity.
Service	Validity	Validity for offering functionality as a service chosen from enumerated type Validity.
Internal Use	Validity	Validity for using the package interally chosen from enumerated type Validity.
ID	Integer	Identification for the package as an integer. For example unique identifier used by organization.
Ownership	OwnershipType	Ownership of the package chosen from enumated type OwnershipType. For example, for marking the package as owned by the author or third party.

Table A.1: *Tagged values of LicensedPackage*

B. APPENDIX: CASE STUDIES

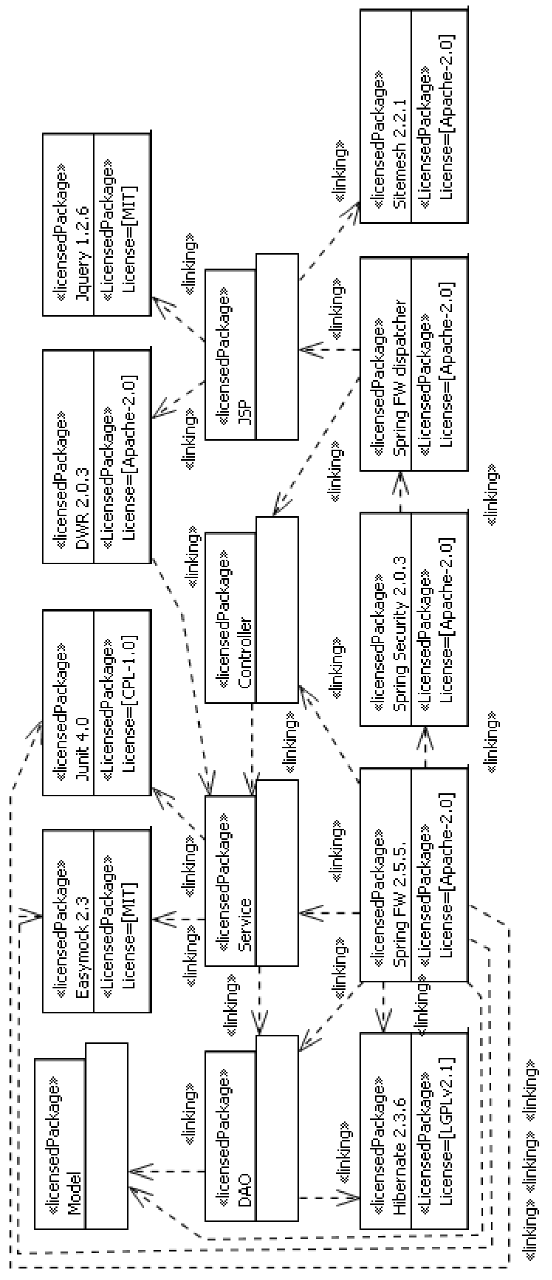

«licensedPackage» Jquery 1.2.6
«LicensedPackage» License=[MIT]
«licensedPackage» DWR 2.0.3
«LicensedPackage» License=[Apache-2.0]
«licensedPackage» Junit 4.0
«LicensedPackage» License=[CPL-1.0]
«licensedPackage» Easymock 2.3
«LicensedPackage» License=[MIT]
«licensedPackage» Model
«licensedPackage» JSP
«licensedPackage» Controller
«licensedPackage» Service
«licensedPackage» DAO
«licensedPackage» Sitemesh 2.2.1
«LicensedPackage» License=[Apache-2.0]
«licensedPackage» Spring FW dispatcher
«LicensedPackage» License=[Apache-2.0]
«licensedPackage» Spring Security 2.0.3
«LicensedPackage» License=[Apache-2.0]
«licensedPackage» Spring FW 2.5.5.
«LicensedPackage» License=[Apache-2.0]
«licensedPackage» Hibernate 2.3.6
«LicensedPackage» License=[LGPLv2.1]
«linking»

Figure B.1: *HOT case study*

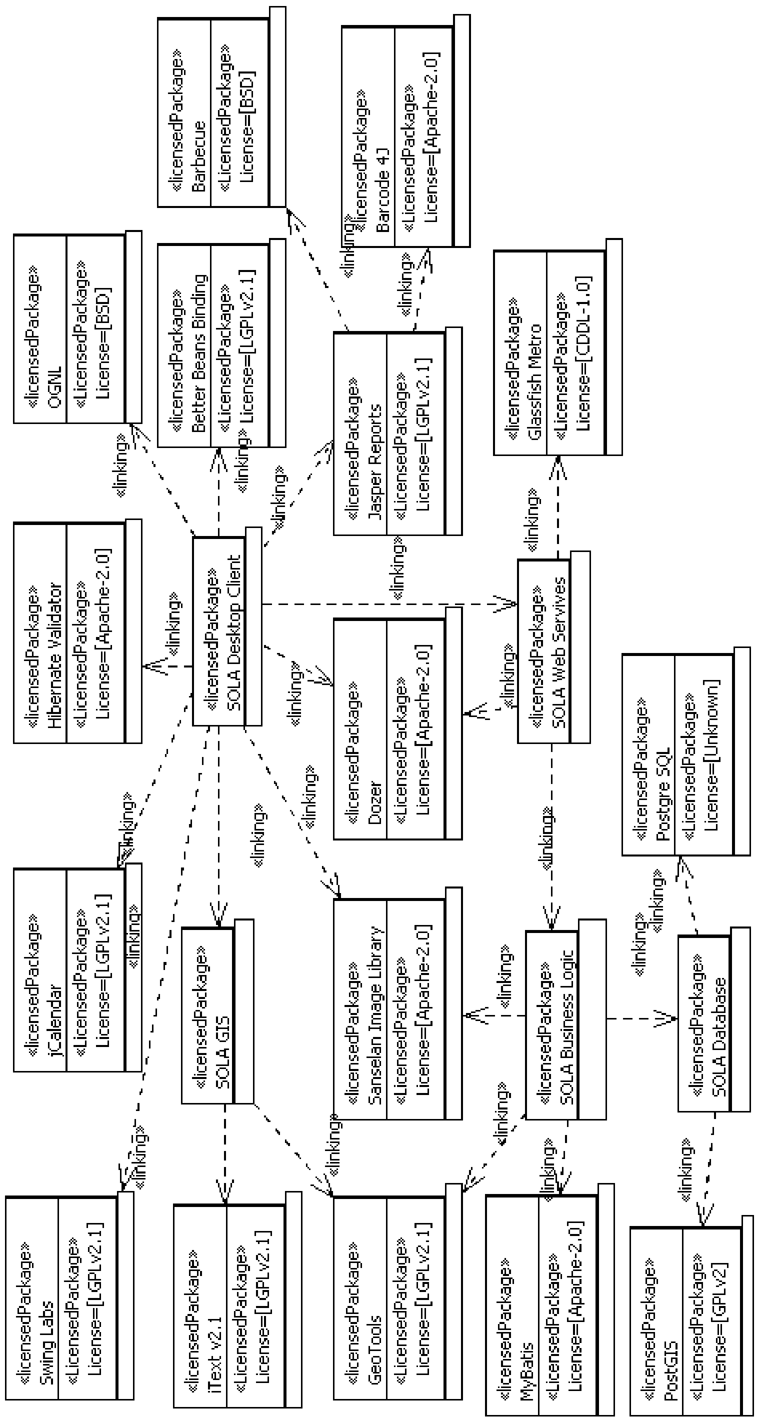

«licensedPackage» Barbecue «LicensedPackage» License=[BSD]
«licensedPackage» Barcode 4J «LicensedPackage» License=[Apache-2.0]
«licensedPackage» OGNL «LicensedPackage» License=[BSD]
«licensedPackage» Better Beans Binding «LicensedPackage» License=[LGPLv2.1]
«licensedPackage» Jasper Reports «LicensedPackage» License=[LGPLv2.1]
«licensedPackage» Glassfish Metro «LicensedPackage» License=[CDDL-1.0]
«licensedPackage» Hibernate Validator «LicensedPackage» License=[Apache-2.0]
«licensedPackage» SOLA Desktop Client
«licensedPackage» Dozer «LicensedPackage» License=[Apache-2.0]
«licensedPackage» SOLA Web Servives
«licensedPackage» Postgre SQL «LicensedPackage» License=[Unknown]
«licensedPackage» jCalendar «LicensedPackage» License=[LGPLv2.1]
«licensedPackage» SOLA GIS
«licensedPackage» Sanselan Image Library «LicensedPackage» License=[Apache-2.0]
«licensedPackage» SOLA Business Logic
«licensedPackage» SOLA Database
«licensedPackage» Swing Labs «LicensedPackage» License=[LGPLv2.1]
«licensedPackage» iText v2.1 «LicensedPackage» License=[LGPLv2.1]
«licensedPackage» GeoTools «LicensedPackage» License=[LGPLv2.1]
«licensedPackage» MyBatis «LicensedPackage» License=[Apache-2.0]
«licensedPackage» PostGIS «LicensedPackage» License=[GPLv2]
«linking»

Figure B.2: *SOLA case study*