Oswaldo: A Semantic Web Enabled Approach for Identifying Open Source License Violations

Christopher J. Forbes

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science (Software Engineering) at
Concordia University
Montréal, Québec, Canada

August 2018

© Christopher J. Forbes 2018

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Christopher J. Forbes

Entitled: Oswaldo: A Semantic Web Enabled Approach for Identifying Open Source License Violations

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair

Dr. J. Yang

______________________________ Examiner

Dr. Gregory Butler

______________________________ Examiner

Dr. Joey Paquet

______________________________ Supervisor

Dr. Jürgen Rilling

Approved by: ______________________________

Dr. Volker Haarslev, Graduate Program Director

August 29, 2018 ______________________________

Dr. Amir Asif, Dean, Faculty of Engineering and Computer Science

Abstract

Oswaldo: A Semantic Web Enabled Approach for Identifying Open Source License Violations

Christopher J. Forbes

Open source license violations are numerous, multifaceted, and pose significant risk to developers and companies in the form of litigation, sometimes resulting in millions in dollars in damages or settlements. Free/Libre and Open Source Licenses utilize copyright law and are written in legalese, which is often outside the scope of a developer's expertise. Software Engineers commit violations of these licenses' terms and conditions easily and often unknowingly. Consequently, increased knowledge, better tools, and sound processes to detect and prevent license violations are extremely important. This work is an investigation in the types of potential license violations that are committed, through direct and transitive dependency hierarchies in hundreds of thousands of real-world software projects. This thesis contributes a novel approach, entitled Oswaldo, that defines and detects three types of license conflicts: Type 1 Simple Violation, Type 2 Embedded Violations, Type 3 Compound Violations. Unidirectional compatibility/incompatibility relationships of major licenses are modelled. Ontologies and Linked Data are advantageously exploited to detect transitive violation Types 2 and 3, as well as the direct violation Type 1. This thesis also reports initial evaluations of these three types of license violations found in the Maven repository.

Keywords: license violation, compatibility, incompatibility, transitive dependency, free/libre and open source software, semantic web, linked data, ontology, knowledge repository.

Résumé

Oswaldo: une approche basée sur le Web sémantique pour identifier les violations de licences à code source ouvert

Christopher J. Forbes

Les violations des licences libres et à code source ouvert (*Free/Libre and Open Source Licenses*) sont nombreuses, multiformes et représentent un risque important pour les développeurs et les entreprises sous forme de litiges, entrainant parfois des millions de dollars en dédommagements ou en règlements hors cours. Ces licences utilisent la loi sur les droits d'auteur qui sont rédigés dans un jargon juridique, mais aussi dépassent souvent les compétences des ingénieurs logiciels. Il leur est donc faciles de commettre des violations des conditions, et ce de manière inconsciente. Par conséquent, une connaissance accrue, de meilleurs outils et des processus solides pour détecter et prévenir les violations de licences sont extrêmement importants. Ce travail est une enquête sur les types de violations potentielles qui sont commises, à travers des hiérarchies de dépendance directes et transitives dans des centaines de milliers de projets logiciels réels. Ce mémoire apporte une nouvelle approche, intitulée Oswaldo, qui définit et détecte trois types de conflits de licence : violation simple de type 1, violation embarquée de type 2, violation composée de type 3. Les relations de compatibilité/incompatibilité unidirectionnelles des licences majeures sont modélisées. Les ontologies et les données liées (*Linked Data*) sont avantageusement exploitées pour détecter les violations transitives de types 2 et 3, ainsi que la violation directe de type 1. Ce mémoire rapporte finalement les premières évaluations de ces trois types de violations de licence trouvées dans le dépôt Maven.

Mots clés : violation de licence, compatibilité, incompatibilité, dépendance transitive, logiciel libre et à code source ouvert, web sémantique, données liées, web des données, ontologie, référentiel de connaissances.

Acknowledgements

Prof. Dr. Rilling, Dr. Keivanloo und künftiger Dr. Eghan: danke vielmals.

Table of Contents

Chapter 1 .... **1**

1.1 Introduction **1**

1.2 Motivation **2**

1.2.1 Motivating Examples 3

1.2.2 Research Statements 6

1.3 Original Contribution **7**

1.3.1 Where's Oswaldo? Potential Use Cases 8

Chapter 2 .... **10**

2.1 Background **10**

2.1.1 The Protection of Intellectual Property and the Inception of Open Source 10

2.1.2 Open Source Software and License Violations 14

2.1.3 Ontologies and the Semantic Web 16

2.1.4 Modelling Source Code Using the Semantic Web 21

2.1.5 Formal Concept Analysis 23

Chapter 3 .... **26**

3.1 Methodology **26**

3.1.1 Preliminary Study 26

3.1.2 Heterogeneous License Cohabitation 30

3.1.3 License Compatibility and Compliance 31

3.1.4 Types of Violations 34

3.1.5 The Semantic Web and Linked Data 36

Chapter 4 .... **37**

4.1 Implementation **37**

4.1.1 Ontological Compatibility 37

4.1.2 Generating Triples 40

4.1.3 Queries 41

4.1.4 Rules 46

4.1.5 Global Analysis 48

Chapter 5 .... **49**

5.1 Results and Evaluation **49**

5.1.1 Experimental Setting 49
5.1.2 Detected Violations: Trends & Influences 50
5.1.3 Type 1 Simple Violations 51
5.1.4 Type 2 Embedded Violations 53
5.1.5 Type 3 Compound Violations 55
5.1.6 Evaluating Actual (and Notional) License Violations 59
5.1.7 Findings 67
5.1.8 Threats to Validity 68
5.1.9 Evaluation Summary 69

Chapter 6 70

6.1 Related Work 70

6.1.1 Code Clone Detection 70
6.1.2 License Violation Detection Tools and Approaches 71

Chapter 7 75

7.1 Conclusion 75
7.2 Future Work 76
7.3 Publications 79
7.4 Bibliography 80
7.5 Appendices 88

7.5.1 TripleConstructor.java 88
7.5.2 Type1Analysis.swift 91
7.5.3 Type2Analysis.swift 93
7.5.4 Type3Analysis.swift 95

List of Figures

Figure 1.1 4
Figure 1.2 5
Figure 2.1 17
Figure 2.2 24
Figure 2.3 24
Figure 2.4 25
Figure 3.1 26
Figure 3.2 28
Figure 3.3 29
Figure 3.4 33
Figure 3.5 33
Figure 3.6 34
Figure 3.7 34
Figure 3.8 34
Figure 4.1 37
Figure 4.2 40
Figure 4.3 42
Figure 4.4 43
Figure 4.5 45
Figure 4.6 46
Figure 4.7 47
Figure 4.8 47
Figure 4.9 48
Figure 5.1 51
Figure 5.2 53
Figure 5.3 55
Figure 5.4 56
Figure 5.5 58
Figure 5.6 63

Figure 5.7 64
Figure 5.8 65
Figure 6.1 72

Glossary

AGPL	Affero General Public License (versions 1 and 2) or GNU Affero General Public License (version 3), created by Affero, Inc. (versions 1 and 2) and the Free Software Foundation (version 3) respectively.
Apache	Refers to the Apache License, created by the Apache Software Foundation
Artistic	Refers to the Artistic License, created by the Perl Foundation
BSD	Refers to the BSD License (both 2-clause and 3-clause variants), originally used for the Berkeley Software Distribution
Code clone	The reproduction (including slight derivation) of one or more lines of code
CPL	Common Public License
Derivative work	Also known as derived work, is a creative work (text, picture, film, source code, etc.) that is a copy, modification, or extension of another creative work
EPL	Eclipse Public License
EUPL	European Union Public License
GPL	GNU General Public License, created by the Free Software Foundation
Incompatibility	See Violation
Inconsistency	See Violation
LGPL	GNU Lesser General Public License, created by the Free Software Foundation

Linked Data	A data store that uses formalized ontologies. Data is stored as RDF triples (also known as facts) in a triplestore. Triplestores can be queried against using a SPARQL query.
MARKOS	The Market for Open Source Software
MIT	Refers to the MIT License, created by the Massachusetts Institute of Technology
MPL	Mozilla Public License
OSS / FLOSS / FOSS	Free/Libre and Open Source Software
Oswaldo	Describes this research's approach to find open source license violations. The name roughly originates from the phrase "On using the Semantic Web to Automate License violation Detection in Open source software."
OWL	Web Ontology Language
POM	Project Object Model, an XML file that describes a software project
Proprietary	Closed source software, developed in private
RDBMS	Relational Database Management System
RDF	Resource Description Framework
Reasoner	Refers to a Semantic Web reasoner, which is used in a triplestore to infer new facts from existing facts, based on queries, rules, and/or ontologies
SBSON	Software Build System Ontology is an ontology and Linked Data repository that models software projects, their dependencies, and other build system information based on information contained in POM files from the Apache Maven software repository
SDK	Software Development Kit

Semantic Web	An umbrella term for a group of standards and technologies that allows the sharing of data across the Internet, akin to how the World Wide Web is a group of standards and technologies for sharing documents
SPARQL	A recursive acronym which stands for SPARQL Protocol and RDF Query Language. It is a query language, similar to SQL, but for querying triplestores.
SWRL	Semantic Web Rule Language
Transitive	Refers to a fact that is implied or calculated from other facts, rules, queries, and/or ontologies
Triple	A piece of data represented by three entitles: subject, predicate, object. I.e. "María, is, happy." Also known as a fact.
Triplestore	The Linked Data database that holds triples and is query-able
URI	Universal Resource Identifier, similar to a Uniform Resource Location (URL)
Violation	A breach of a contractual agreement in copyright law that has previously been defined by a FLOSS license

Chapter 1

1.1 Introduction

The role of a software engineer is threefold: develop software on time, on budget, and with quality [1]. Developers are trained to plan and create software to meet both computer science and project management perspectives. Meeting these often-conflicting goals and balancing these viewpoints is already time consuming and difficult. Yet in everyday practice, it quickly becomes clear there are other perspectives — legal and community perspectives, that must also be taken into account, when producing software. Little if any time of a software engineer's training is spent studying to understand software from a community or legal perspective [2]. When these perspectives begin to impact our software projects, developers often feel unprepared and overwhelmed. This lack of attention may be due to unawareness of the field, or simply because it is difficult to navigate through large projects to determine precedence, ownership, and copyright.

Intellectual property and its violations repeatedly made business and technology headlines over the past years. Most notably, the widely popular Android mobile operating system garnered much attention in the mainstream press for alleged copyright violations of open source Java code, brought forth in a lawsuit approaching nine billion dollars in damages [3]. Copyright violations involving Open Source Software (OSS) are more common than previously thought, which gives credence to the complexity and scope of such problems. It is imperative for any company, researcher, end-user, or software engineer touching OSS to comprehend these issues as the potential consequences are dire.

The fact that community perspectives and judicial issues are missing from standard training does not mean that these subjects have escaped the notice of software developers [2]. On the contrary, OSS arose explicitly to foster a culture of sharing and collaboration among software developers, while reusing existing source code [4]. As OSS has exploded in popularity, whole communities and sets of rules have sprung up around not only the practical questions of what actually constitutes source code sharing, but also around ethical and legal issues of fairness, freedom, power structures,

autonomy, ownership, collectivity, and the like. These questions and their attempted answers have been enshrined in copyright notices, also known as Open Source Licenses. Open source licenses grant legal rights to a developer (or other stakeholders) to read, modify, and share the source code of a project. Legal literacy is well out of the scope of a modern Software Engineering curriculum, and yet is demanded in the day-to-day practice of sharing code.

Knowing what constitutes an appropriate use — and misuse! — of OSS is lacking in programmers today. Even though open source and free software is well over 30 years old now, there are still many myths and misconceptions surrounding the correct use of OSS. If license compliance is not achieved, a license violation occurs. As its name implies, a license violation is an infraction of the law. Its consequences can be as banal as a friendly email from a project owner reminding the programmer how to properly comply with the terms of the license [5], or they can entail major lawsuits involving millions in sales of currently shipping products with settlements of princely undisclosed sums [6].

Unsurprisingly, given the potential seriousness of a license violation, many approaches and tools have been developed [3], [7], [8], [9], [10] to detect possible license incompatibilities. However, no two software projects are the same, each software license has varying terms and conditions, and the usage of OSS is more nuanced and far-reaching than can be dealt with by any of the existing tools. This continues to be true in spite of the recent push for intercompatibility between licenses; measures taken so far have slightly alleviated but by no means resolved the problem.

The objective of this this dissertation, therefore, is to develop a new technique to support the detection of license violations that will take into account the complexity and dependencies of real projects where often multiple licenses are involved.

1.2 Motivation

The discipline of Software Engineering has always concerned itself with producing quality software on time and on budget. Part of making high quality software includes not only rigorous

creation processes, but also the surrounding issues of legality with regards to copyright laws and Open Source Software.

As copyright law varies from nation-state to nation-state, lawyers and court systems concern themselves with specific cases. As software engineers, we can also help fellow developers and project stakeholders avoid problems of litigation, by checking the license compatibility before we begin the coding process and integration of libraries. However, when the responsibility for this preventative action falls on the shoulders of an unprepared programmer, it can cause unnecessary strain.

Writing code is not always a smooth exercise. Developers have to focus on many simultaneous factors, such as: business-critical hot-fixes, departmental deadlines, performance reviews, and nonsensical bugs. Developers do not need the additional pressure of combing through terms and conditions written in legalese in order to make decisions outside of their professional field of expertise. However, the potential negative effects of one hasty wrong decision can be disastrous. The aim in this present research is to introduce an approach which will ease the creation and maintenance processes for software practitioners by guiding them through the identification and analysis of copyright violations in their projects.

In addition to constructing a tool of practical value, we also discuss the complexities and limitations of Free/Libre and Open Source Software (FLOSS), including the blended use of open source licenses. The whole open source movement is an intriguing and ingenious method to institute fairness among communities of developers, and ultimately to preserve within the industry the basic human desire to help another person. My hope is to continue this tradition with this research.

1.2.1 Motivating Examples

Many open source license violation lawsuits have made waves in the news over the years [3], [6], [11], [12]. These monetarily and emotionally high-stakes cases merit research activities because of the incredibly high legal consequences of being found in violation of copyright and the very steep monetary penalties that follow. For example, the popular maker of network routers Cisco

was sued by the Free Software Foundation (FSF) [11]. Like most of these license violation cases, both parties settled outside of court. Not only were monetary damages (of an unknown sum) paid by Cisco to the FSF but also part of the agreement Cisco was required to modify its software development practices and publish the infringing source code in question.

This is not the only case of its kind. BusyBox has successfully brought litigation and settled multiple cases of license infringement, most notably with Verizon in 2008 [6]. Currently there is an ongoing case, started in 2015 in Hamburg, Germany, between an independent open source contributor Christoph Hellwig and VMWare [12]. There is also the infamous case of Oracle vs. Google with its disputed "nine lines of code" and nine billion dollars in damages, recently resolved in 2016 [3]. More research surrounding these license violations and their prevention is a main driver in this research.

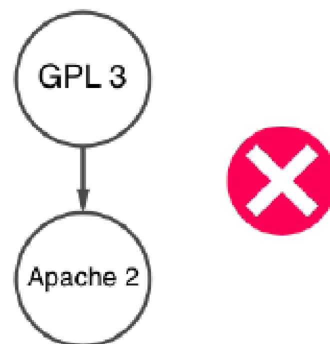


Figure 1.1 An example dependency hierarchy where a simple program is published using the Apache 2 open source license. This program has one dependency released under the GPL 3 license. This relationship is forbidden by the GPL 3. Thus, a violation occurs. Note: the arrow merely represents the "flow" of code from the dependency GPL 3 project which is then used in the Apache 2 project.

In the research field of open source license violations, there has been little exploration into source code license violations specifically using the Semantic Web and Linked Data. Linked Data is particularly suited to this use case: the hierarchy of libraries each released under their respective license terms yet used together to form the various components of a program. This hierarchy is not easily captured and analyzed using traditional development toolchains, but the Semantic Web

provides well-suited techniques (knowledge modelling and knowledge inference) to represent and tactfully analyze this problem space. For example, Figure 1.1 shows a very simple dependency hierarchy, yet a more complex dependency hierarchy (in either breadth, width, or both) with multiple levels of dependencies (like in Figure 1.2) is well suited to Linked Data.

Furthermore in this field of research, there has been inadequate analysis of transitive license violations (only one such study [13] was conducted). While direct dependencies are somewhat easier to detect (e.g. [7], [8]), finding indirect non-compatible relationships is much more challenging, since these transitive relationships are often the result of third-party libraries or components imported by dependency managers or package management systems. Figure 1.2 shows a transitive dependency where the top-most project was imported due to the middle dependency. In this research we take advantage of the Semantic Web and its reasoning services, which provides a flexible approach for modelling (ontologies) and inference of such transitive dependencies (SPARQL queries, SWRL rules)

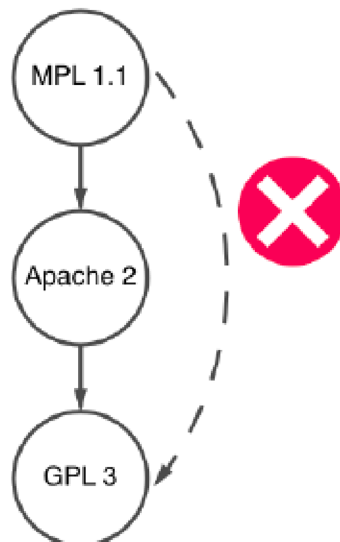


Figure 1.2 An example project dependency hierarchy which includes a transitive violation. There exists a transitive relationship that is forbidden; that is, the MPL 1.1 project cannot be used in the GPL 3 project.

1.2.2 Research Statements

As the hierarchy of imports increases — both horizontally and vertically — in a software project, the likelihood of a potential license violation increases. Manual inspection of license terms is time consuming and error prone. Therefore, we posit the following primary research statement:

> *An approach will be developed based upon a flexible license modelling method that can detect both potential direct and transitive license violations.*

The acceptance criteria used to validate the above research statement will be as follows; if the technique detects both first and second order violations, then the research statement is considered provable, otherwise it will be considered disproved. (A second order violation will be fully described in the methodology section. Put simply, a direct import of one library to another is a first order connection. A second order connection is a library that imports a library.) Thus, by proving or disproving this research statement the main research goal of helping software engineers avoid license violations when using OSS licenses through the use of the Semantic Web will be fulfilled.

A secondary research statement is as follows:

> *Furthermore, we expect that the most common type of violation will be a directly dependent Type 1 Simple Violation, the second most common a transitive Type 2 Embedded, and the third a transitive Type 3 Compound Violation.*

The thought processes behind such a statement are as follows. Type 1: Developers are generally unaware, are not legal experts, and have other pressing concerns, such as finishing a project on time. Type 2: While it is easy to manually check the immediate project's license compatibility, it is hard to manually corroborate a dependency's dependencies. Type 3: Similar to the reasoning for type 2, it is easy to manually check immediate first-order dependencies, but difficult to manually check transitive dependencies.

These research statements will be addressed over the course of this memoir.

1.3 Original Contribution

In order to make a novel contribution to the field, we set out exploring various facets of the domain. The following summarizes some of the main contributions of the presented research.

Using Linked Data, we published a large "Internet-scale" data set of source code online called SeCold (Source code ECOsystem Linked Data) [14]. As part of this work I was exposed to "big data," the Semantic Web, and the creation of Linked Data data sets. The published data set consisted of 1.5 billion triples and 18 000 open source projects. This data set was publicly accessible through a SPARQL query endpoint (as well as data dump files) to provide the research community with Internet-scale clone and code search tools. We also introduced Doppel-Code, a plug-in for the Eclipse IDE, which visualized and ranked clone results based on clone impact factors [15]. Unlike other clone detection tools, Doppel-Code leveraged "both local and global clone information, and therefore its application can be found beyond typical clone detection applications, such as prioritizing of bug fixes as part of the triage problem [15]."

Next, we investigated the legal implications of publishing 18 000 projects together in one database [16]. This lead to questioning the cohabitation of licenses. In investigating, we considered the "'Generational Limitation' that affirms: 'derivative works [must] be distributed on the same terms' as the original license [16] [17]." As part of this study, we found that copyright law does not broadly apply to data sets [18]. Thus with SeCold, we had not committed any license violations.

Subsequently, we focused on the use (or rather the reuse) of code clones. A code clone is generally one line of code that is reused. They are categorized into a handful of types (imaginatively named Types 1 to 4) differing in whitespace, variable names, additions or deletions of statements, and syntax (while remaining semantically alike). We built another Eclipse plug-in for detecting code clones, known as source code similarity search or more succinctly as clone search. This plug-in was essentially an online clone search engine which is powered by SeCold as a back-end. As SeCold was a Linked Data repository, we straightforwardly enriched the data set by adding a new clone ontology [19]. Seen in the broad context of our research, this tool was an exploration of code reuse on a micro level.

Finally, as part of the Ambient Software Research Group at Concordia University, we expanded upon SeCold by showcasing how it can be used as a platform, to benefit the Mining Software Repositories (MSR) research community [20]. The first use case is Linked Data-based Fact Browsing. "SeCold facilitates both browsing of source code (i.e. Linked Data-based Source Code Browsing) and retrieval of related facts (e.g. similar source code) using the Linked Data [20]." The second use case is License Violation Mining. SeCold could surface license information on the source-code line level due to the integrating of various information silos into one Linked Data platform. License violation detection between code clones could be performed with a SPARQL query.

An initial implementation of a license violation detection tool was created as described in the Preliminary Study section. Later on, a novel approach entitled Oswaldo[1] was developed, to fulfil the primary research statement.

1.3.1 Where's Oswaldo? Potential Use Cases

Why make (another) license violation detection tool? Oswaldo is a query endpoint based on an underlying ontology that can be utilized by various stakeholders in the software development industry. Oswaldo allows for the reuse of predefined queries or by writing new custom queries to define new types of license violations. In addition, this Semantic Web-based knowledge modelling approach provides the ability to extend the existing license model as new knowledge becomes available, furthering the flexibility and longevity of such a technique.

Developers can use Oswaldo to query for who uses their library as a dependency to detect potential license violations. For example, the BusyBox authors whose open source project has a history of violations (by the likes of Best Buy, JVS, Samsung, Westinghouse [21]) could identify additional violations and email those organizations to ask them to comply with the terms of the license.

---

[1] Oswaldo is loosely derived from the phrase "On using the Semantic Web to Automate License violation Detection in Open source software."

Legal counsels of an organization could use Oswaldo and its analysis results to verify whether their software contains any potential license violations, thus assessing and reducing the firm's current exposure to litigation risk. Also, being conscientious of which license combinations cause the most violations aids lawyers in shaping company-wide legal policies, such as an internal list of open source licenses approved for use and disapproved for use, thus proactively avoiding future lawsuits.

Oswaldo is useful for researchers, since it consolidates and publicizes the license ontology and the Maven data set through a SPARQL endpoint.

Chapter 2

2.1 Background

2.1.1 The Protection of Intellectual Property and the Inception of Open Source

The Open Source Software (OSS) movement sprung up in the late 90s as an alternative to proprietary copyrighted software. OSS is defined as applying a copyright license to a piece of software where the license adheres to set of principles including the right to change, modify, and redistribute publically the source code. OSS is a branch of the Free/Libre and Open Source Software (FLOSS) movement, also known as 'copyleft,' F/OSS, or FOSS, which is more fervent and ideologically driven. Since its inception in 1997, FLOSS and OSS have become extremely popular to the point where a majority of software projects released today contains some form of OSS [22].

Proprietary software is privately developed software that remains the protected intellectual property of the developer. This means that only the binary is sold or distributed to the end user: the source code, like all creative works, is automatically subject to copyright law (unless otherwise specified) [17]. Most countries with strong software industries, i.e. U.S., U.K., China, Canada, Germany, France, etc., have similar copyright legal protections in place [17]. The purpose of these laws has remained essentially the same since it was first developed with the introduction of the printing press in the UK. Authors wanted assurance of compensation for their creative work, rather than revenue going solely to the publisher who copied their work using the printing press [17].

Interestingly, copyright law does not cover the idea itself, merely the expression of the idea. Instead, patent law creates legal protections for ideas. A simple example of this is that a company like MySpace, for example, could have patented the *idea* of 'online social networking.' However, since they did not do so, Facebook was able to make a new creative work (a web application) based on the MySpace concept. Copyright laws, by contrast, would apply to the *source code* of MySpace

and Facebook, so neither could use each other's code unless they were given permission. Thus, "the implementation of the source code can be copyrighted since it is the expression of the idea," rather than the idea itself [16].

Copyright laws were created with the goal of a fair distribution of profits. Paradoxically, Open Source Software stemmed from a similar goal: fairness of use and modification. When big corporations first began invoking copyright law to withhold source code, independent software developers felt that was a violation of the existing sharing culture. According to the norms of that culture, source code should be made available to independent programmers so that they would be able to modify software according to their communities' needs (often a small bug fix) [23]. Withholding source code made this customisation difficult, expensive or impossible.

The obvious response to proprietary licensing was to release a work into the public domain, which relinquishes all copyright, effectively publishing the work in the open. However, this response did not satisfy the independent developers who had opposed proprietary licensing, because modifications to public domain source code remained permissible, and those modifications (subject to copyright) may then be kept private.

Chiefly incensed among the independents who objected to the copyrighting was Richard Stallman. Stallman pioneered the copyleft and Free Software movement by creating the GNU General Public License[2] to not only preserve the communal spirit of source code sharing from what he called "software hoarding [24]," but also from malicious actors. *Free* does not refer to cost, but to *Freedom*. Likewise, Stallman prefers the term Free Software to Open Source because he views access to and modification of source code as an issue of justice: a check-and-balance and fundamental human right, like Freedom of the Press, or Right of Assembly.[3] His copyleft licensing was a creative twist on (and ironic use of) copyright law. Free/Libre and Open Source Software cleverly uses copyright laws to ensure that the work remains accessible rather than protected as

---

[2] This thesis uses the American English spelling of the noun "license" rather than the British spelling of "licence" simply because the former is more widespread in the literature.

[3] I had the opportunity to hear Mr. Stallman speak at McGill University in May 2017.

private property. Rather than releasing the work into the public domain (which would permit copyrighting modifications), copyleft ensures that all modifications to the work will remain public, and that, in consequence, “no proprietor can exclusively exploit a creative work [16].”

Although Stallman’s copyleft initiative was embraced by many developers, and in fact remains a popular form (if not the most popular form) of opening sourcing, other developers objected to its “moralizing and confrontational” tone [25]. In 1998, developers who preferred a more business-friendly approach to Open Source Software formed an organisation called the Open Source Initiative and agreed upon an Open Source Definition as a basis for determining whether a software licence could be labelled with the open source certification mark [26]. The details of the Open Source Definition will be described in some detail in the pages that follow.

2.1.1.1 Principles of Open Source Software

As defined by the Open Source Initiative, Open Source Software is more than simple access to source code online. True Open Source Software conforms to a set of principles intended to actively *encourage* sharing and reuse [27]. Those principles are the following:

1. **Free redistribution** of the software is permitted
2. **Source code** must be publicly available and accessible
3. **Derivative works** (modifications to the source code) must be allowed
4. **Integrity of the author's source code** e.g. derivative works may require a different name
5. **No discrimination against persons or groups**
6. **No discrimination against fields of endeavor**
7. **Distribution of license** means the license rights apply to whomever the software is given
8. **License must not be specific to a product**
9. **License must not restrict other software** that is distributed alongside the licensed software
10. **License must be technology-neutral**

However open source is not just a set of philosophical principles. It is firmly rooted in copyright law. “Because an open source license is unilateral, each grant is granted provided a set of

conditions are satisfied; if one of such conditions is violated, then the grant is not given by the US Court of Appeals for the Federal Circuit in Jacobsen v. Katzer [22] [28]."

Table 2.1 Ten common open source licenses and their traits.

License	Requires Attribution[4]	Requires Public Source Code for Derivative Works	Requires Same License for Derivative Works
Apache 2	Yes	No	No
Artistic 2	Yes	No	No
BSD[5] — Berkeley Software Distribution License	Yes	No	No
EPL 1 — Eclipse Public License	Yes	Yes	Yes
GPL 2 — GNU General Public License	Yes	Yes	Yes
GPL 3	Yes	Yes	Yes
LGPL 2.1 — GNU Lesser General Public License	Yes	Yes	Yes
LGPL 3	Yes	Yes	Yes
MIT — Massachusetts Institute of Technology License	Yes	No	No
MPL 2 — Mozilla Public License	Yes	Yes	Yes

2.1.1.2 Types of Open Source Licenses

Open source licenses generally fall into two categories: restrictive and permissive. An OSS license is a legal instrument that allows the creative work (source code) to be used, modified and/or shared

[4] "Requires Attribution" generally means posting in your software's credits, the title of the OSS project, and a copy of its license (with the optional but karma-filled posting of: the author, and a link to the project's website).

[5] BSD can refer to a handful of variations on the same license. For the purposes of this paper the common 2- and 3-clause variants are used.

under defined terms and conditions [24] [27]. Restrictive licenses (colloquially referred to as copyleft licenses) require derivative works to be licensed under the same terms. A derivative work is defined as any work that stems or is adapted from the original work [13]. An example of a restrictive license is the GPL 3. Permissive licenses on the other hand have fewer requirements on derivative works; for example, the MIT License only requires author attribution and reproducing the license with the disturbed software. The Table 2.1 above lists ten most frequent licenses with pertinent features summarized [29].

2.1.2 Open Source Software and License Violations

2.1.2.1 Software Libraries, Repositories, Package Managers, Compliance, and Incompatibility

Today source code is shared in many forms over the Internet. Probably the most common form of sharing code is through the use of a software library. Libraries have become popular because they are self-contained and perform a set of functions. Multiple libraries can be grouped together in a build repository to form an SDK (Software Development Kit) or a complete subsystem. Usually a library has one OSS license applied to it. This is often true for SDKs as well but not exclusively so. With the proliferation of build repositories and package managers, such as RubyGems [30], Maven [31], or CocoaPods [32], which make downloading and importing a library into your project as trivial as a one-click affair, many projects and SDKs include libraries from various authors with a plethora of OSS licenses. One can picture this scenario as a horizontal increase in the project's dependency graph. Furthermore, one library can use another library, leading to hierarchies of libraries (seen as vertical increase in the dependency hierarchy). All of these libraries' licenses must be compatible with each other, or depending on the license at the very least with its direct neighbours in the hierarchy.

When incompatible licenses are used together, a license violation occurs. A license violation is defined as *"the act of making use of a [licensed] work in a way that violates the rights expressed by the original creator* [33]*."* That is, not following the legal terms and conditions set out in the

open source license.[6] Software authors who commit a licence violation open themselves to the possibility of being sued. Sometimes this risk can amount to millions of dollars, as in the recent case of Oracle v. Google [3].

It should be noted that even though the term *license violation* is used throughout this thesis, a definitive violation is only determined as such by judge or jury. Consequently, *potential* is the operative word when discussing license violations. As many countries' judicial systems are based on the concept of precedent, lawyers can generally determine what constitutes an actual violation based on prior rulings. Precedent is "a previous case or legal decision that may be or (binding precedent) must be followed in subsequent similar cases [34]."

2.1.2.2 Code Clones

Licence violations can occur at different granularity levels, because incompatibilities can occur not only on the macro scale of libraries, but also on the micro scale of source code fragments. Popular websites such as GitHub (which contains millions of repositories of source code, both open and proprietary) and Stack Overflow (a peer-to-peer self-help developer discussion forum) provide valuable but potentially hazardous repositories of ready-made source code for developers. Close attention must be paid when copying and pasting a few lines of source code into a project, so as to not create any license conflicts. Lines that are copied and pasted in this way are commonly known as "code clones". Code clones are created from two code fragments which is "any sequence of code lines (with or without comments) [35]." A code clone is formally defined: "a code fragment CF2 is a clone of another code fragment CF1 if they are similar by some given definition of similarity, that is, f(CF1) = f(CF2) where f is the similarity function [35]." Although code clones are easily discovered by many existing tools [35], [19], they pose many threats because they are not easily separable from the rest of the software project. Thus, they may easily create license violations. Fortunately, there are existing tools available to software developers to mitigate the risks of license violations of this kind, some of which are described in the related work section.

---

[6] The term *license compliance* refers to the act of following the conditions of the license. Whereas *license compatibility* generally signifies whether two licenses can be used together while maintaining compliance with each license's terms.

2.1.2.3 License Proliferation

The creation of more and more open source licenses is known as License Proliferation [28] [36]. Since each license has its own unique wording and terms, which may or may not be compatible with the terms used in another license, the proliferation of licenses greatly increases the likelihood that licence violations will occur. For example, the government of Québec created not one, but a family of three (3!) licenses, "Licence Libre du Québec" (LiLiQ), with Permission, Reciprocity, and Strong Reciprocity variants [37]. The GNU Foundation has been trying to combat the legal uncertainties of license proliferation by having their legal experts study popular licenses and maintain a list of GPL-compatible licenses. This is published online and is an excellent resource [38].

2.1.3 Ontologies and the Semantic Web

Humans make sense of the world around by classifying the various plants, animals, and objects of their environment into groups. This activity leads to the development of formal classification systems that identify the members of a class and model the relationships among these members. Such systems are known as ontologies (literally, studies of being). Thomas Gruber distinguished between taxonomies and ontologies as follows: "Ontologies are often equated with taxonomic hierarchies of classes, class definitions, and the subsumption relation, but ontologies need not be limited to these forms [39]." That is, an ontology is a representation of the world, but not an exhaustive one.

Any concept classification can be modelled by an ontology. An example of this is language instruction. Language teachers of Indo-European languages have traditionally taught their students how to classify the words in a sentence according to their "part of speech" and their grammatical function within the sentence: verbs (passive, active, etc.), nouns (including subject and object of the sentence), adjectives, and adverbs. Take the phrase "María plants habanero peppers." María, the subject-noun, acts upon the object-noun, habanero peppers, by planting them, the verb. Modelling the concept of a sentence using the subject-verb-object ontology is a useful tool to help students interpret more complex grammatical structures. Similarly in software, computer scientists model the real world to tackle complex problems such as voice recognition.

2.1.3.1 Representing an Ontology

More formally, an ontology can be represented as a graph, with nodes and links. In this data structure, the nodes represent concepts and the links denote their attributes. Attributes are usually types, properties, or relationships among the concepts, such as "is a", "instance of", or "related to". The below Figure 2.1 illustrates the relationship María has to the peppers. María (concept) plants (attribute) peppers (concept).

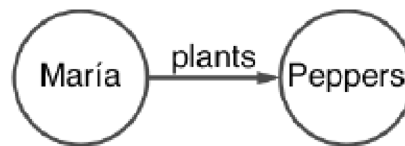


Figure 2.1 A graph representation of an example ontology.

This graph representation of the relationship is also known as a semantic network. These networks form the basis of the Semantic Web. The Semantic Web is defined as "a common framework that allows data to be shared and reused across application, enterprise, and community boundaries [40]." The Semantic Web employs ontologies for knowledge representation, which facilitates the sharing of data through common concepts. (Ontologies with equivalent concepts can easily be linked together using a "sameAs" relation, unlike traditional relational databases.)

2.1.3.2 Resource Description Framework and Triples

Initially created by the World Wide Web Consortium (W3C) as a metadata model for the web, Resource Description Framework (RDF) has now broadened in scope to include more general conceptual descriptions of data and web resources, to help facilitate data merging and evolving schemas [41]. RDF creates a basis for the Semantic Web. "The RDF abstract syntax is a set of triples, called the RDF graph [41]." Put otherwise, a "triple" is used to represent data. The following formalism denotes a triple:

$$\tau = < S, P, O >$$

A triple is characterized by a graph-based representation with an ontological structure of subject, predicate, object [41]. The subject and object are represented as nodes in the graph (María and Peppers in Figure 2.1), and the predicate is the link between the nodes (plants in the same example). Each component in an RDF triple is backed by a URI (Uniform Resource Identifier), apart from the object which can either be a URI or a literal value, e.g. string, integer. The use of URIs allows for concrete referencing and simple processing. This is a powerful way to describe relationships between two objects. Continuing with our example we could have the following RDF triple:

$$\tau = < \texttt{http://example.com/garden\#Maria}, \\ \texttt{http://example.com/garden\#Plants}, \\ \texttt{http://example.com/garden\#Habanero_Peppers} >$$

2.1.3.3 Linked Data and Triplestores

The most common approach currently used to model data is a relational database, such as MySQL, Oracle RDBMS, SQLite, which arrange data in tables, columns, and rows. Relational databases do not explicitly use ontologies, but rather they use entity-relationships. Although this approach to data storage and retrieval has many advantages, it also has many shortcomings, which include rigid schemas, data migrations, sharding, data silos, etc. [16]

Linked Data is a form of data modelling that uses ontologies to overcome many of the limitations of traditional table-based databases [16]. Linked Data was invented in 2006 by Tim Berners-Lee (the inventor of the World Wide Web) under the umbrella of the Semantic Web movement at the W3C [16] to share data between computers as easily as the web makes sharing information between humans.

Linked Data builds upon RDF triples to represent data facts. These triples are stored in a database known as a triplestore, and are used to build the object graph. The key feature of Linked Data is that these object graphs can be queried against using a SPARQL query at a query endpoint. SPARQL is an acronym for "SPARQL Protocol and RDF Query Language" [42] which is similar to an SQL (Structured Query Language) query. A query endpoint is a standardized web service available at a URI where an end-user can submit their SQARQL query, run it, and have structured

Linked Data results returned. Moreover, because of the object graph, implied transitive relations can be found between nodes even though two nodes are not directly linked. This is a powerful feature when linking together two online Linked Data sets that share the same nodes or same relations. For example, we know that María plants habanero peppers, tomatoes, and epazote[7]. However, we further know that she only plants vegetables because habanero peppers, tomatoes, and epazote each have an "is a" relation pointing to the "vegetable" node[8]. This combination of graph representation and online sharing easily connects otherwise siloed data — allowing new queries and furthermore new results to be performed and found.

2.1.3.4 Advantages of Linked Data

There are seven facets of Linked Data that set it apart from traditional relational databases [20]. These are:

1. **Extensible Data Schema** — a predefined schema is not required, unlike JSON, XML, and relational database tables. A vocabulary set is used to "model concepts (e.g. Bug, Commit, Variable Name, and Java Class) and relations (e.g. hasAuthor) in the domain of discourse. At any time, the model can be extended by adding new terms. Moreover, it is possible to have various revisions of the model at the same time [20]."
2. **Feasible and Scalable Reasoning** — complex logic and computationally intensive reasoning is not mandated when using Linked Data (unlike the Semantic Web) [43]. Yet transitive reasoning is still possible.
3. **Online** — Each object in the graph is represented by a URI, and thus is dereferenceable. "That is, anybody on the Web should be able to access facts related to the target entity using its URL via HTTP [20]."
4. **Human Accessible** — A human readable format must be accessible from a modern web browser when navigating to the URI. Also, relations and related facts must be shown at this URI.

---

[7] Epazote leaves, used in Guatemalan and Mexican cuisine, are added to a dish, akin to cilantro.

[8] The forgiving culinary definition of "vegetable" is used, which includes common fruits such as peppers and tomatoes.

5. **Accessible to Software** — A software application can use the URI to retrieve relations and related facts in a standard format i.e. XML or JSON.
6. **Queryable** — Anyone (machine or human) is able to query the online repository and have the matching resultant facts returned
7. **Integrable** — Facts can be easily integrated together to find transitive results. Since each fact has its own online URI, not only can intra-data set queries be performed, but more powerful inter-data sets queries can be executed just as easily. Thus, a *federation* of data sets can be created. This capability to be simply agglomerated is very powerful [20].

In short, Linked Data models real world concepts, stores data, and enables complex queries with new insightful results. The Semantic Web stack optimizes for different use cases than relational databases. As a general rule, relational databases are better for tabular data, while Linked Data is a better choice if one is representing complex data models with non-static queries.

2.1.3.5 Reasoners, Transitive Relationships, SPARQL Queries, SWRL Rules

Reasoners are programs that deduce implicit relationships from explicit triples. These implicit relationships are otherwise known as transitive relationships. Reasoners use Description Logic (DL) language, such as OWL-DL Ontologies (Web Ontology Language–Description Logic), to restrict relationships and thus infer new relationships. Unlike RDF which only permits parent-child relationships, OWL-DL can add many other constructs such as: unions, local scope, cardinality restrictions, inverse relationships, disjointed relationships, etc. For example, male squash blossoms are *disjoint* from female squash blossoms. The power of reasoners is that they make use of these constructs and existing triples to provide de facto implied information for little work.

Reasoners are directly important to the research described in the paper because the reasoner's deduced transitive relationships will be used to model the various characteristics of open source licenses. For example, certain licenses are incompatible with each other (parent-child hierarchical relationship, inverse relationships). Yet there are other licenses that merely cannot be used in the same project with each other (local scope, disjointed, etcetera).

In order to tell a reasoner to execute a specific query, an SQL-inspired query language was devised, called SPARQL (pronounced “sparkle”) [42]. For comprehension’s sake one can think of the following analogy: SPARQL is to triplestores as SQL is to relational databases. In addition to queries, rules can be specified. Instead of executing query after query to find license violations, one can use the Semantic Web Rule Language (SWRL) to write rules to automatically identify the license violations [44].

2.1.4 Modelling Source Code Using the Semantic Web

Source code was first modelled using the Semantic Web by Keivanloo et al. in 2012, which has been previously discussed in the Original Contribution section.

Later on in 2014, the MARKOS (MARKet for Open Source) project [45] resulted from a Europe-wide effort to model licenses using Semantic Web technologies. The MARKOS project models many different aspects of a software project including “functional, structural and licensing aspects of the software” but for the purposes of this research we are exclusively focused on the license ontology (which is further described in the methodology section).

In 2018, Eghan et al. created the Ontology-based Trustworthiness Assessment Model (OntTAM) [46] which builds upon various other ontologies including the Software Engineering Evolvable Quality Assessment Metamodel (SE-EQUAM) [47], the Security Vulnerability Analysis Framework (SV-AF) ontology [48], the Software Build Systems Ontology (SBSON) that models build repositories, etc. [46] Here, the authors created an ontology which combines source code facts from all 1500 projects in Maven. The authors’ main goal was to develop a metric of trustworthiness to compare various open source dependencies, and then use this metric in the OntTAM ontology and corresponding data set. The metric encompasses both security vulnerabilities and open source license violations (and is based on the research presented in this thesis). Please refer to Figure 4.2 for more detail of the OntTAM and SBSON ontologies.

2.1.4.1 MARKOS Types of License Permissions/Violations

In order to find possible license violations, one must outline permissions that describe the various use cases of two licenses, and of these use cases, which licenses are not allowed to be used together. Happily, the MARKOS ontology [45] (in OWL format) includes various use cases as part of the class "LicenseTerm" which has a subclass of "Permission". There are six permissions listed in Table 2.2 below.

Table 2.2 Permissions defined in the MARKOS ontology.

Permission	Description
Adaptation	An OSS license allows the original creative work to be adapted and modified.
Distribution	One can publicly distribute the source code.
LibraryUsageWithout Reciprocity	Depending on whether the license is copyleft or 'permissive,' it may require reciprocity. Reciprocity is defined as "the practice of exchanging things with others for mutual benefit, especially privileges granted by one country or organization to another [34]." When applied to licenses, reciprocity means the source code from a derivative project must be released under the same license as the derived project in order for both libraries to be used together. E.g. María makes changes to a seed planting season calendar app licensed under the GPL 3. Because of the reciprocity requirement in the GPL 3, she must release her changes under the same license when she posts her app online.
PatentGrant	Some source code algorithms or processes are patented, and the author agrees to grant permission to any downstream user of the source code.
Reproduction	One is allowed to reproduce or make copies of the source code.
Sublicensing	A user of this source code is permitted (or not) to sublicense the code to anther license.

From the permissions described above (Table 2.2), it is clear that each permission can be violated. Therefore, there are generally six types of violations that can occur. Reciprocity is important to this research because its context is straightforwardly captured by an ontology and easily relatable

to the Maven repository of projects. The reciprocity requirement mainly influences (but is not the sole requirement for) the definition of license compatibility demarcated later on in this thesis.

Beyond reciprocity, some of the other permissions are harder to detect violations because they are violated outside of the realm of a Software Engineering context. For example, the authors of BusyBox sued Samsung in 2009 [21] and settled in 2010 [49] because Samsung was using BusyBox's FLOSS project without publicly publishing the source code (when distributing the software with their hardware). This is a violation of the distribution term of the GPL 2 (which would equate to the Distribution permission in the MARKOS ontology). This was only found by manually checking the physical product (in this case a Samsung television) and verifying that the FLOSS was indeed running on the TV hardware. We do not (yet!) have an automated way of testing all the physical products in the world. Therefore, in creating a definition of license violation, we must combine multiple permissions that are feasible to determine. These permissions provide a basis to construct definitions of compatibility, incompatibility, and license violations, which will be further described in the Methodology section.

2.1.5 Formal Concept Analysis

Formal Concept Analysis (FCA) was popularized by Rudolf Wille, a mathematician at the Technische Universität Darmstadt, Germany, starting in 1982 with his seminal work "Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts" [50]. The goal of FCA is to categorize objects and their attributes into relationships and hierarchies. The following notation can be used:

$$B(G, M, I)$$

Where:

B stands for *Begriff* (Concept)
G for *Gegenstand* (Object)
M for *Merkmal* (Attribute)
I for *Inzidenzrelation* (Relation, or incidence relation)

The combination of an object, attribute, and relation will produce a concept. A reader will note that G, M, and I, form a triple. This information can be represented in tabular form (Figure 2.2).

Context (G, M, I)

Objects (G)	mild	medium	hot	orange	red	green
Habanero			×	×		
Scotch Bonnet			×	×	×	
Ruqutu			×	×	×	
Malagueta			×		×	
Piquín		×			×	
Serrano		×			×	
Jalapeño	×					×
Poblano	×					×

Figure 2.2 An example context table showcasing hot chili peppers with their approximate spiciness and colour.

Not only can the concept lattice be represented as a table, but also as a graph lattice (Figure 2.3):

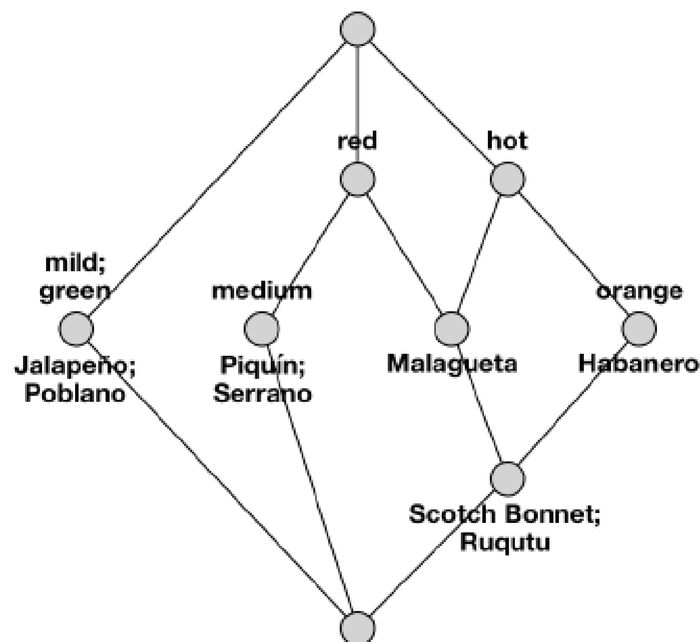


Figure 2.3 Example concept lattice depicting hot peppers.

Concept 1	({}, {mild, medium, hot, orange, red, green})
Concept 2	({Scotch Bonnet, Ruqutu}, {hot, orange, red})
Concept 3	({Habanero, Scotch Bonnet, Ruqutu}, {hot, orange})
Concept 4	({Scotch Bonnet, Ruqutu, Malagueta}, {hot, red})
Concept 5	({Piquín, Serrano}, {medium, red})
Concept 6	({Jalapeño, Poblano}, {mild, green})
Concept 7	({Habanero, Scotch Bonnet, Ruqutu, Malagueta}, {hot})
Concept 8	({Scotch Bonnet, Ruqutu, Malagueta, Piquín, Serrano}, {red})
Concept 9	({Habanero, Scotch Bonnet, Ruqutu, Malagueta, Piquín, Serrano, Jalapeño, Poblano}, {})

Figure 2.4 Example calculated concepts table.

From FCA's resultant tables and graph the reader can easily infer: (1) which objects share attributes amongst themselves. (2) how this sharing naturally leads to the identification of sets of similar objects (concepts) (shown in Figure 2.4). (3) how, from sets of objects, relationships and hierarchies are easily recognized. The above reasons denote the major advantages of FCA as a useful data analysis tool.

Chapter 3

3.1 Methodology

As detailed in the literature review, there exist a fair number of techniques and approaches to model data, classify it, and detect license violations. This chapter introduces Oswaldo which employs some of these methods, as well as outlines the basis for their use. The process is outlined in Figure 3.1.

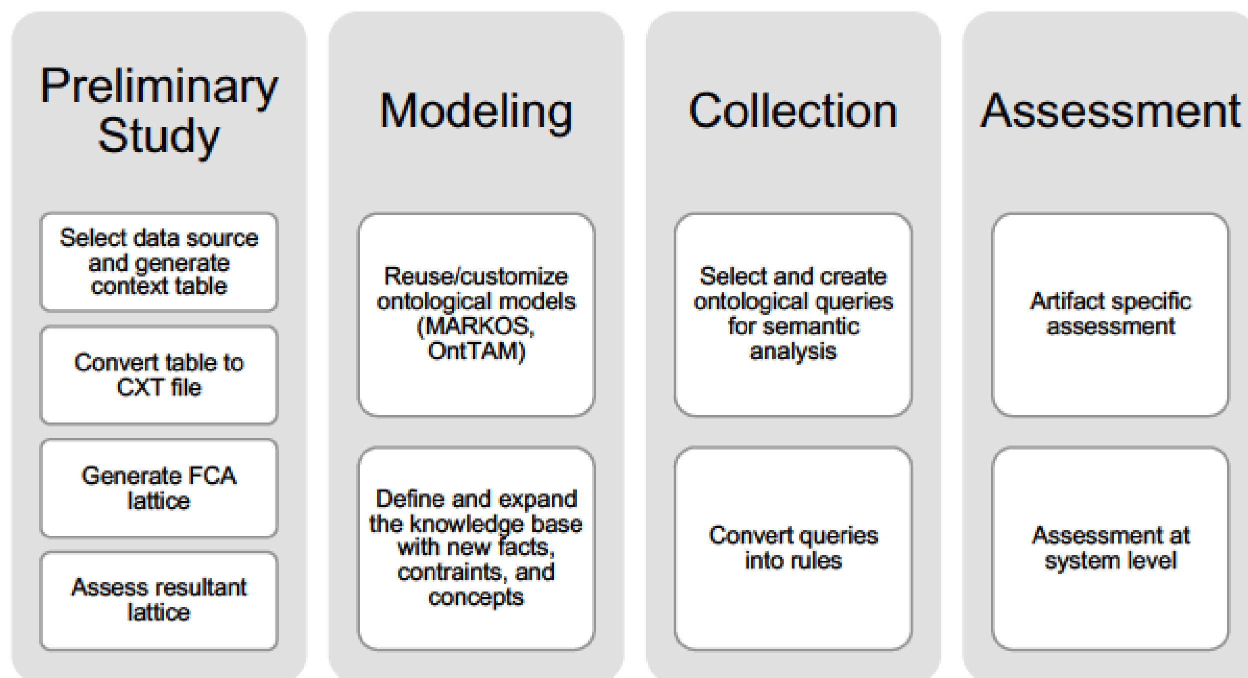


Figure 3.1 Oswaldo methodological process.

3.1.1 Preliminary Study

Part of the duties of a software developer include selecting and integrating various open source projects and libraries and integrating them into a developing project. Questions that a developer may pose to herself during the selection process can include: "Can I use this open source library

with the current ones I'm using?"; "What are the differences between the various licenses?"; "What features define them?" [29], [36], [38] To answer these questions, a more apt tool is required.

A FCA lattice would be an ideal approach for the visualization of such information. If one were to create a formal concept of the all the most common licenses, and each license's attributes, a concept lattice could be generated. A developer could find her currently in-use license, and then attempt to trace her finger to the new project's license that she would like to use. If there is no connection between the two, then a license violation would result if she used both licenses together. Conversely if there was a connection in the lattice, then in general she should be able to use both licenses together.

The one caveat of this approach is the attribute "same license". On rare occasions, a license will require derivative works to be licensed only using its parent's license. This difficulty is increasingly becoming more well-known as the license proliferation problem [28], [36]. Many licenses have explicitly stated compatibility with each other to circumvent this caveat.

For the development of a license compatibility tool, we take advantage of data from ChooseALicense.com which is compiled and maintained by GitHub employees on behalf of the software community. Herewithin, GitHub engineers compiled the most commonly used licenses, as well as grouped their terms and conditions into several attributes [29]. Although this data is presented in tabular form, it remains difficult for a human being to derive meaning from the table, and thus apply its findings to their own project. FCA is thoroughly suited to overcome this difficulty.

As part of our approach we converted the GitHub data to a context table for the most pertinent and popular licenses (Figure 3.2 below).

Attributes (M)

Objects (G)	Commercial use	Distribution	Modification	Patent use	Private use	Disclose source	License and copyright notice	Network use is distribution	Same license	State changes	Liability	Trademark use	Warranty
AGPL 3	×	×	×	×	×	×	×	×	×	×	×		×
Apache 2	×	×	×	×	×		×			×	×	×	×
Artistic 2	×	×	×	×	×		×			×	×	×	×
BSD 2	×	×	×		×		×				×		×
BSD 3	×	×	×		×		×				×		×
CC BY 4	×	×	×		×		×			×	×	×	×
CC BY-SA 4	×	×	×		×		×		×	×	×	×	×
CC0 1	×	×	×		×						×	×	×
Eclipse 1	×	×	×	×	×	×	×		×		×		×
EUPL 1.1	×	×	×	×	×	×	×	×	×	×	×	×	×
GPL 2	×	×	×		×	×	×		×	×	×		×
GPL 3	×	×	×	×	×	×	×		×	×	×		×
LGPL 2.1	×	×	×		×	×	×		×	×	×		×
LGPL 3	×	×	×	×	×	×	×		×	×	×		×
MIT	×	×	×		×		×				×		×
MPL 2	×	×	×	×	×	×	×		×		×	×	×
MS-PL	×	×	×	×	×		×					×	×
MS-RL	×	×	×	×	×	×	×		×			×	×
PostgreSQL	×	×	×		×		×				×		×
The Unlicense	×	×	×		×						×		×

Figure 3.2 Context Table of Various Licenses and their Attributes, based on the data provided by GitHub [29].

From the context table, we manually converted the data into CXT format. This format is a simple text file to represent the context table, which is commonly used by FCA tools. With the CXT file, we downloaded an FCA tool called RubyFCA [51], [52]. RubyFCA is a command line tool which uses GraphViz and implements one of the original FCA algorithms by Bernhard Ganter [53]. Thus, from the CXT file we generated the corresponding lattice (Figure 3.3):

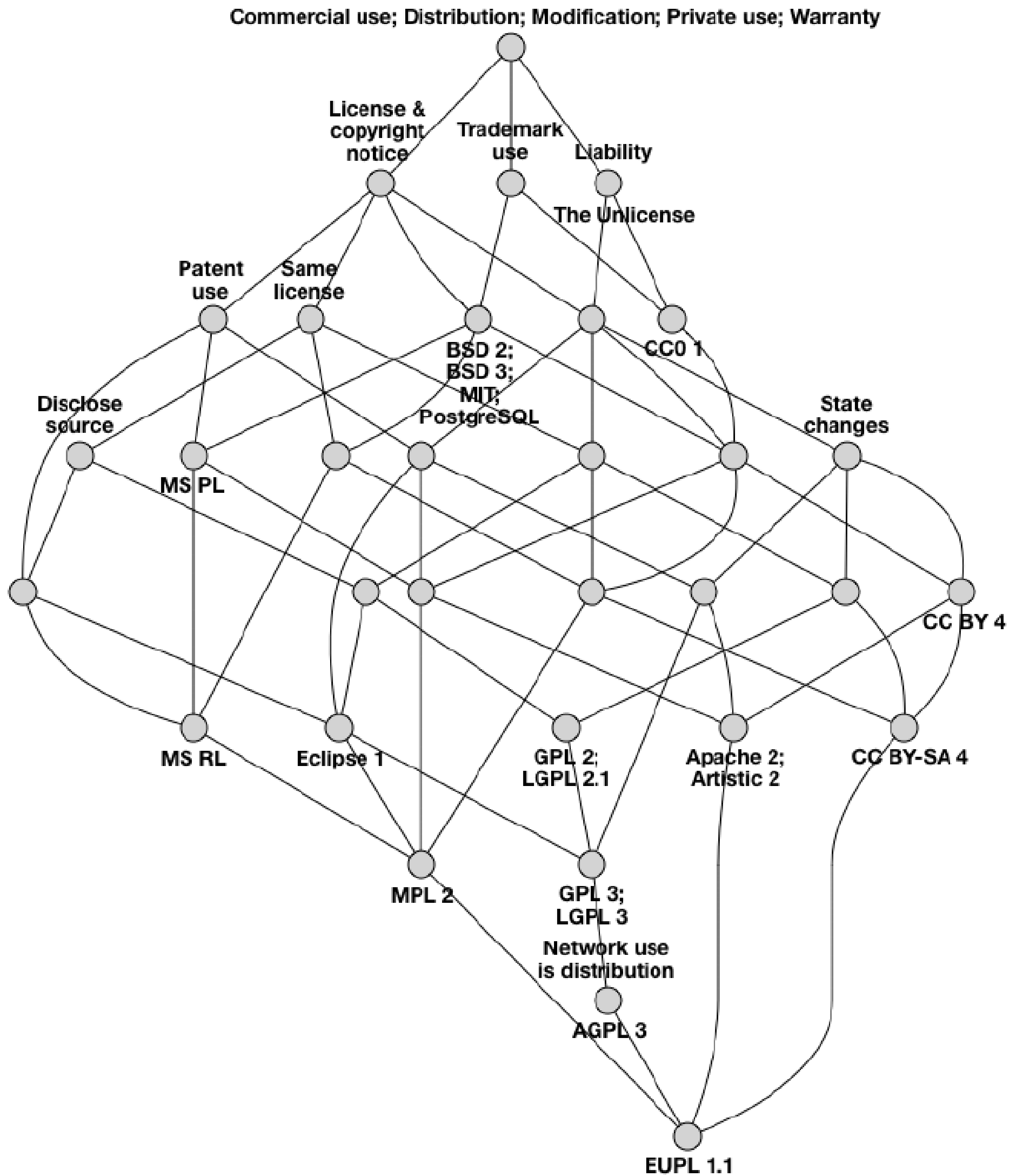


Figure 3.3 Concept lattice depicting various licenses and their attributes.

This lattice shows the groupings within the licenses based on the hierarchy of objects and attributes. For example, licenses in the bottom left are all strongly copyleft due to those nodes

sharing the same license attribute. The lattice allows one to trace various licenses or groups of licenses to their attributes, which is intuitively done with one's finger. As a tool, this lattice is valuable for a variety of stakeholders. For project authors newly minting a project as open source, the lattice can help them compare the features of competing licenses. For an organization's legal counsel, this could help them in recommending and approving licenses to avoid and use based on the organization's specific needs.

Using FCA, developers can obtain answers to two of the three common questions when selecting an open source library: "What are the differences between the various licenses?" and "What features define them?" However, more investigation is needed in order to fully answer the third common question: "Can I use this license with that license?" Answering this question requires detecting current license violations in real-world projects.

3.1.2 Heterogeneous License Cohabitation

Existing license violation detection tools focus on finding violations through the use of source code comparison [9], code clone detection techniques [3], sentence matching [8], assembler and binary analysis [10]. These methods typically provide narrow solutions in terms of identifying license violations; what they lack is *wider context*.

First, the issue of licensing compatibility is not just between two licenses, but between all licenses in use in a current project! Many projects have multiple dependencies with varying licenses. All terms and conditions of all licenses in use must all simultaneously be adhered to.

Second, how the project is *used* within the dependency hierarchy affects whether a license is compatible with another license. Put simply, compatibility is not bidirectional. This context will be discussed in detail below.

Even though licenses themselves have many terms and conditions, all major licenses do follow a general set of principles (as described in the literature review [27]). A better approach that takes into account the overall context is needed.

3.1.3 License Compatibility and Compliance

The legal compatibility of two or more licenses while adhering to their dense terms and conditions is a complex topic. It must be stated upfront that compatibility between two licenses is technically outside the scope of Software Engineering, and instead it is best to defer to legal experts and FLOSS advocacy organizations for specific questions [38]. As discussed in the literature review for example, the GNU Foundation outlines which OSS licenses are compatible with the latest version of the GPL. As the interplay of licenses is a multifaceted topic, this research seeks to help shed light on these issues, and thus give meaning to this investigation.

Compatibility is a rather precarious choice of wording to describe the ability to use two licenses together. The Oxford dictionary of English defines compatible as: "able to exist or occur together without problems or conflict [34]." This simple definition of compatible does not fully capture the relationship of using two licenses together. The word "compatible" is a puzzling choice because it has connotations of bidirectional agreement. (The phrase "mutually compatible" comes to mind.) License compatibility, despite being poorly named, is not bidirectional, but unidirectional.

How do two projects exist together, each with their own license? One imported project exists *inside* a second actively developed project, which is normally distributed under the second project's license. For example, the Apache 2 license is compatible with the GPL 3 license in that "Apache 2 software can therefore be included in GPLv3 projects, because the GPLv3 license accepts our software into GPLv3 works. However, GPLv3 software cannot be included in Apache projects [54]." This is important to note because it makes preventing a license violation that much more difficult for a Software Engineer. It also means that the required ontology to model these relationships is that much more complex.

Some organizations have used the terms "upstream compatibility" and "downstream compatibility" to denote the unidirectionality of compatibility between two licenses [55]. However, since the term "compatible" is widespread among various OSS communities online, we continue its use by clarifying below.

3.1.3.1 Definitions of Compatible and Incompatible

Compatible: can be copy-pasted or imported into (including linking).

There is one limitation to the above definition for this thesis, which mainly concerns the LGPL.[9] "compatibleWith" when used in conjunction with the LGPL means a library can be linked, **but source code cannot copied and pasted into the project**.

The LGPL defines linking as "a work that uses the library" either statically (compile-time) or dynamically (run-time) where the *source* code of both projects is not combined. Each project is an "independent work that stands by itself, and includes no source code from [the other]." This separation does not apply to *compiled* code [56].

The reason for considering this limitation to the definition of "compatible" is simple. The LGPL was created so that GPL projects could be relicensed under the LGPL and thus the project could be linked to. (The L originally stood for "Library" because the general use case for the LGPL is linking the project as a library [57]). We made this exception, otherwise many incorrect results would turn up if the LGPL is considered incompatible with other licenses. Please see section 7.2 Future Work for a discussion on how to mitigate the limitations of this definition.

---

[9] LGPL and AGPL are both slight exceptions to the definition of compatible. An AGPL 3-licensed project can be imported into a GPL 3 project. But AGPL 3 code cannot be copy-pasted into a GPL 3 project. This is noted in the ontology used to gather the data.

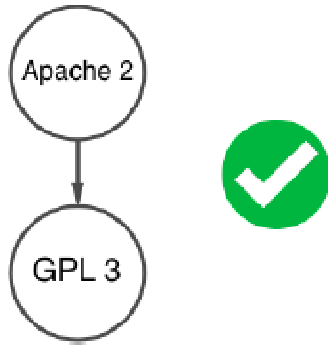


Figure 3.4 Representation of the triple: "Apache 2 compatibleWith GPL 3"

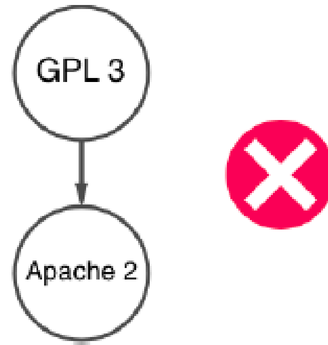


Figure 3.5 Representation of the triple: "GPL 3 incompatibleWith Apache 2"

For the purposes of this dissertation we use the following notation to denote the project dependency hierarchy, that is, the flow of code from one project into another (which does not imply compatibility nor incompatibility):

Apache 2 ▸ GPL 3

The MARKOS ontology attempts to take the compatibility concept into account with its "compatibleWith" property on the individual license. However, the "compatibleWith" property is incomplete for most licenses included in the ontology. While our ontology is based on the MARKOS ontology and the "compatibileWith" property, we had to expand it in order to capture the different types and scopes of license violations. (Figure 3.4 shows an example project hierarchy with compatible licenses.) As well, the "incompatibleWith" property was added (Figure 3.5 shows an example incompatible relationship) because as a rule, ontologies follow the open world assumption; that is, simply because one fact is true, does not mean the opposite is false. Therefore, we present a second definition.

Incompatible: cannot be copy-pasted nor imported into (nor linked to).

The discussion of defining what is compatible and incompatible may seem like an exercise in hearing one's voice echo in an ivory tower, but this clarification is in fact very pragmatic for a

software developer. The downstream 'consuming' project — its author! — is accountable for complying with all the terms of all its dependent licenses. This includes any embedded or hidden dependencies. Accordingly, since we have defined what is compatible and incompatible, we must hence define license violations.

3.1.4 Types of Violations

In the context of our research, three types of OSS license violations are distinguished and identified. The following definitions are put forward:

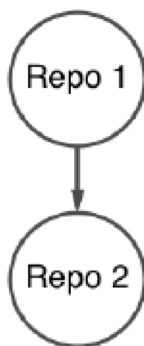


Figure 3.6 Type 1 Simple Violation

Figure 3.7 Type 2 Embedded Violation

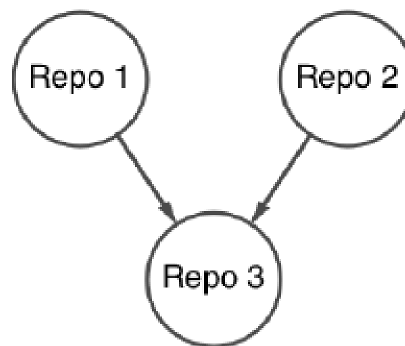


Figure 3.8 Type 3 Compound Violation

Type 1 Simple Violation — When two licenses are directly used together (e.g. one imports the other), yet one of the license's terms disallow this combination. (Figure 3.6)

Type 2 Embedded Violation — Three licenses are used together in linear fashion; the first license is a dependency of the second where the second itself is a dependency of the third. The first and

third licenses have incompatible terms. Thus, a transitive license violation potentially occurs. (Figure 3.7)

Type 3 Compound Violation — Three licenses make up the dependency graph in a triangular formation, where the third project has two dependencies. These two dependent projects are incompatible, hence the potential license violation. (Figure 3.8)

Some key details about each type should be noted. Type 1 is the easiest violation to prevent. This "easiness" is not why Type 1 is named "simple." The *usage* of the dependency is simple. The direct use of the dependency means the developer is aware of the use of that dependency, and can read through both licenses terms to determine compatibility (or even search online for "license1 license2 compatible").

Type 2 and Type 3 are both referred to as transitive because the interaction of their dependencies is not direct but indirect. For Type 2, the developer of the third project is likely unaware of the violation occurring in its dependency hierarchy. The developer would have to painstakingly search through all of second's project's code to find the first project is used a dependency. Then she would need to read through all three licenses to determine whether a violation has occurred. The crux of Type 3 is that a developer can verify that her first dependency is compatible with the license of her project, and do the same for her second dependency. Even though the developer thought she completed her due diligence, she has not checked the compatibility *between* her two dependencies.

These two transitive types are particularly relevant to our research. The downstream user of a dependency is highly unlikely to know about this potential violation as it is not only *deep* in the dependency hierarchy, but also *difficult* to manually detect (manual tracing of dependencies is required). Detecting license compatibility is indeed a difficult problem for developers and the technique developed in this research has (like our character María) cultivated new ground to finding these violations. This is exactly why these violation types have been defined.

3.1.5 The Semantic Web and Linked Data

Now that three license violation types have been define, the reader can see the complexity in detecting Types 2 and 3. Semantic Web technologies such as ontologies, reasoners, Linked Data representations, and SPARQL endpoints are well suited to modelling and querying for these transitive types of data [20]. Much less work (when comparing to traditional relational databases) is needed to model the project relationships (because transitive relationships are inferred and queryable). Similarly, the OntTAM project was a natural fit to build upon. Furthermore, Linked Data ontologies are easily extensible and combinable using "equivalent URLs that point to the same entity (i.e. `owl:sameAs`) [14]." As a Linked Data data set, which captures knowledge about build management systems, and based on the metadata from the Maven project, it models 1 849 756 project releases. Additionally, incorporating a SPARQL query endpoint allows developers and researchers to devise and execute their own queries. Furthermore, the query endpoint could be used as an API to support any type of development environment or tool such as an Eclipse plug-in, NetBeans, PHPStore, Android Studio, or Xcode. This internet-scale data set approach is ideal for finding any license violations.

Chapter 4

4.1 Implementation

The process for developing Oswaldo was mostly straightforward (Figure 4.1). To start, we captured and modelled the license compatibility information by extending the MARKOS ontology. Next, we generated the license triples and used them to populate the Linked Data server. After, we created the SPARQL queries and SWRL rules. The final step was to analyze and make sense of the query results using Excel and small programs.

Figure 4.1 Implementation process for creating Oswaldo.

4.1.1 Ontological Compatibility

The MARKOS ontology was used as a basis for license compatibility (and incompatibility), since MARKOS already has a list of commonly used licenses. Each license is listed as an OWL individual with permissions. However, for our purposes of license conflict detection, the ontology needed to be expanded. In this section we have outlined the steps to modify the ontology to our needs.

First, MARKOS contains the following major OSS license: AGPL, Apache, Artistic, BSD, CPL, EPL, EUPL, GPL, LGPL, MIT, MPL. But it is missing some older versions of these licenses: Artistic 1, GPL 1, LGPL 2, MPL 1. Those older versions are of particular interest to license violations because many of the original licenses were made before license proliferation was identified as a problem. In short, they have been added to the ontology.

The ontology does refer to each license by its name. However, in order to satisfy Linked Data requirements, every fact must be uniquely identifiable. Thus there exists an individual annotation "rdfs:seeAlso" with a URI to the license on the web. This would generally be sufficient if everyone linked to the same license URI in their open source project, but of course this is not the case. Variations on the same URI are common, including http vs. https, .txt vs. .html vs. .php, and source domain i.e. gnu.org vs. opensource.org, etc. As such, all of these variants were meticulously added to the ontology.

The next step was to define the compatibility relationships between each license. This process was meticulously completed for 6 licenses (out of 21). ($6 \times 21 = 126$. $126 \div 441 = 29\%$ of the top 21 most common licenses.) These six licenses are: GPL 1, GPL 2, GPL 3, Apache 1, Apache 2, Apache 3. (See Table 4.1 below for the general idea.) This means that over 126 compatibilities were manually verified and populated into the ontology. In fact, more than 126 compatibilities were recorded, but for simplicity's sake, we do not include them because not all compatibilities were recorded for all 21 licenses. I.e. The triples of Apache 1 & 2 into MPL 1, 1.1 & 2 were recorded but not Apache into CPL. This subset of six licenses was chosen because GPL and Apache are consistently some of the most popular licenses online. Additionally, the process of verifying whether the use of two licenses constitute a license violation is a very time consuming. A reputable source had to be found online for each compatibility pair and then the result entered into the ontology using the ontology editor application Protégé [58].

Table 4.1 Tabular demonstration of the data entered into the ontology. This table is not complete. The reader can read the table as follows: "Column importable into Row". E.g. "Apache 1.1 is compatible/importable into MPL 2." This data has been made available online as an OWL file [59].

	AGPL 3	Apache 1	Apache 1.1	...
AGPL 3	✔			
Apache 1	×	✔	✔	...
Apache 1.1	×	✔	✔	...
Apache 2	×	✔	✔	...
Artistic 1				
Artistic 2				
BSD 2-clause				
BSD 3-clause				
CPL 1				
EPL 1				
EUPL 1.1				
GPL 1	×	×	×	...
GPL 2	×	×	×	...
GPL 3	✔	×	×	...
LGPL 2				
LGPL 2.1				
LGPL 3				
MIT				
MPL 1		×	×	
MPL 1.1		×	×	
MPL 2		✔	✔	

4.1.2 Generating Triples

With the license compatibility relationships modelled, the next step was to create some triples from the build ontology. This process was fairly straightforward as we could build from the previous work: the OntTAM/SBSON ontologies and data set [46], outlined in Figure 4.2. The first step was to execute TripleConstructor.java, which takes the license URI from SBSON and compares that to the license URI in MARKOS and generates the triple: "Release, coveringLicense, License", which means "this release is covered by this license." This triple represents the match between the license entity (in MARKOS) and its equivalent Release (in SBSON).

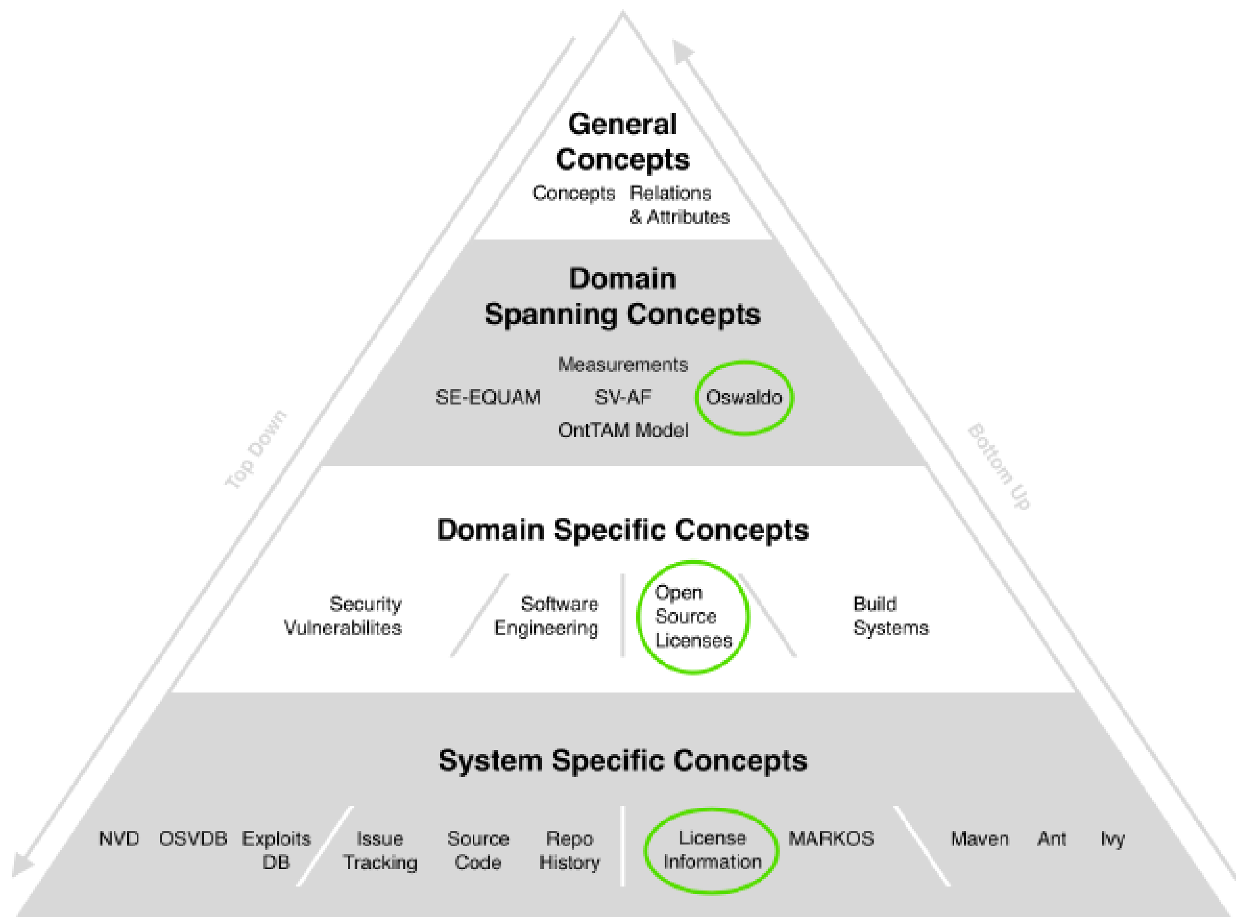


Figure 4.2 The Software Trustworthy Ontology Hierarchy. License Information and MARKOS are combined to make Oswaldo which contributes to this model.

Once the triples are generated, it is a simple process to populate the data set, that is load them into Virtuoso Server [60], by following these steps:

- Copy the MARKOS owl file into the directory ~/virtuoso/dataset
- Run TripleLoader.java file in Eclipse
- Check that the triples are loaded using the SQL query using the interactive SQL command line tool isql: `select * from DB.DBA.load_list;`

The result of this processing step is a populated Linked Data data set, which is ready to be used through a SPARQL endpoint.

4.1.3 Queries

Since the types of license violations have already been defined, the next step was to create SPARQL queries to detect each type of license violation.

Please note the following technical language of the ontology included in the following queries:

- Dependency Target is the dependency being imported into the second project.
- Dependency Source is the project "receiving" the imported code. The reader can think of this from the perspective of a developer, where this is their current *source* code being actively developed, and which is importing third-party code (depicted below in Figure 4.3).

These naming conventions are based on the ones used by the SBSON (Software Build System Ontology) [46].

A further note should be pointed out regarding that the choice of directionality of the arrows in our dependency figures. The arrows show the flow of code and do not show the dependency graph (unlike the SBSON project), because this research is principally concerned with derivative works and what license the third-party code is ultimately disturbed under in a downstream project that uses that third-party code.

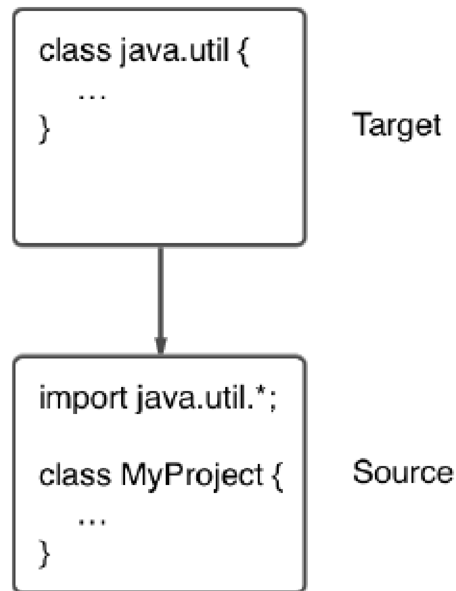


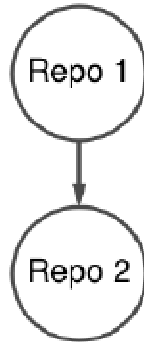
Figure 4.3 Two example classes with an arrow to show the flow of code from the target dependency to the source.

4.1.3.1 Type 1 Simple Violation

Objective:

Find two projects (repositories), where each project has a different license, and one project is imported into the other, and this importing relationship is not allowed by the terms of either license.

Query:




```
PREFIX build:
<http://aseg.cs.concordia.ca/segps/ontologies/domain-
specific/2015/02/build.owl#>
PREFIX markosCopyright:
<http://www.markosproject.eu/ontologies/copyright#>
PREFIX markosLicense:
<http://www.markosproject.eu/ontologies/licenses#>

select distinct *
where {
?link a build:DependencyLink.
?link build:hasDependencyTarget ?repository1.
?link build:hasDependencySource ?repository2.

?repository1 markosLicense:coveringLicense ?license1.
?repository2 markosLicense:coveringLicense ?license2.

?license1 markosCopyright:incompatibleWith ?license2.

FILTER (?license1 != ?license2)
}
```

Figure 4.4 The SPARQL query to detect a potential Type 1 Simple Violation.

Result:

The query will return pairs of projects where the first project is imported into the second project and the license of the first project cannot be imported into the license of the second project; thus each pairing represents a license violation, e.g. repository1, link, repository2, license1, license2.

4.1.3.2 Type 2 Embedded Violation

The next query was written to discover violations between two licenses with one intermediary. It is for Type 2 Embedded violations. Take note of how simple it is to make the query transitive; this is simply achieved using the triple "`?license1 markosCopyright:incompatibleWith ?license3.`"

Objective:

Find three projects (repositories), where each project has a different license, and one project is imported into a second project that is imported into a third project, and this importing hierarchy is not allowed by the terms of the first or third license.

Query:

```
PREFIX build:
<http://aseg.cs.concordia.ca/segps/ontologies/domain-
specific/2015/02/build.owl#>
PREFIX markosCopyright:
<http://www.markosproject.eu/ontologies/copyright#>
PREFIX markosLicense:
<http://www.markosproject.eu/ontologies/licenses#>
PREFIX markos:
<http://www.markosproject.eu/ontologies/oss-licenses#>

select distinct *
where {
?linkA a build:DependencyLink.
?linkA build:hasDependencyTarget ?repository1.
?linkA build:hasDependencySource ?repository2.

?linkB a build:DependencyLink.
?linkB build:hasDependencyTarget ?repository2.
?linkB build:hasDependencySource ?repository3.

?repository1 markosLicense:coveringLicense ?license1.
?repository2 markosLicense:coveringLicense ?license2.
?repository3 markosLicense:coveringLicense ?license3.

?license1 markosCopyright:compatibleWith ?license2.
?license2 markosCopyright:compatibleWith ?license3.
?license1 markosCopyright:incompatibleWith ?license3.
```

```
FILTER (?license1 != ?license2 && ?license1 != ?license3
&& ?license2 != ?license3)
}
```

Figure 4.5 The SPARQL query to detect a potential Type 2 Embedded Violation.

Result:

The query will return trios of projects, where the first project is — but should not be — imported into the third project (through an intermediary second project). Each trio of projects represents a Type 2 license violation. E.g. repository1, linkA, repository2, linkB, repository3, license1, license2, license3.

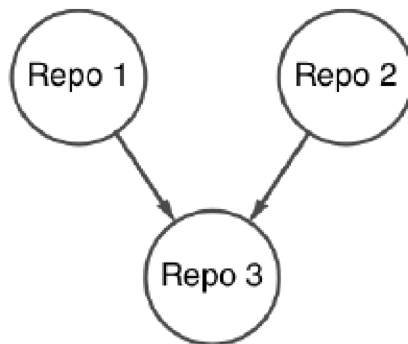
4.1.3.3 Type 3 Compound Violation

The following query captures the third and final license violation: Type 3 Compound.

Objective:

Find three projects (repositories), where each project has a different license, and the first project is imported into the third project as well as the second project is imported into a third project, and this importing hierarchy is not allowed by the terms of the first or second license.

Query:




```
PREFIX build:
<http://aseg.cs.concordia.ca/segps/ontologies/domain-
specific/2015/02/build.owl#>
PREFIX markosCopyright:
<http://www.markosproject.eu/ontologies/copyright#>
PREFIX markosLicense:
<http://www.markosproject.eu/ontologies/licenses#>
PREFIX markos:
<http://www.markosproject.eu/ontologies/oss-licenses#>

select distinct *
where {
```

```
?linkA a build:DependencyLink.
?linkA build:hasDependencyTarget ?repository1.
?linkA build:hasDependencySource ?repository3.

?linkB a build:DependencyLink.
?linkB build:hasDependencyTarget ?repository2.
?linkB build:hasDependencySource ?repository3.

?repository1 markosLicense:coveringLicense ?license1.
?repository2 markosLicense:coveringLicense ?license2.
?repository3 markosLicense:coveringLicense ?license3.

?license1 markosCopyright:compatibleWith ?license3.
?license2 markosCopyright:compatibleWith ?license3.
?license1 markosCopyright:incompatibleWith ?license2.

FILTER (?license1 != ?license2 && ?license1 !=
?license3 && ?license2 != ?license3)
}
```

Figure 4.6 The SPARQL query to detect a potential Type 3 Compound Violation.

Result:

The query will return triplets of projects, where the first project is imported into the third project while the second project is imported into the third but the first and second project should **not be used together**. Every triplet returned from the query signifies a Type 3 license violation, e.g. repository1, linkA, repository2, linkB, repository3, license1, license2, license3.

4.1.4 Rules

Similarly, the SWRL rules for the three violation types are as follows. As a reminder, SWRL rules and SPARQL queries differ in that the SWRL rules permit for automatic inferencing and materialization. Based on a given SWRL rule, the reasoner infers new facts from existing facts. If new facts are inferred, these are automatically added to the triplestore (materialization). For example, if new projects and dependencies are added to the triplestore, the SWRL rules will

automatically generate the corresponding potential license violations facts. However, the Virtuoso Server triplestore that was used in these experiments does not support SWRL rules, which is why these rules are outlined in the figures below (Figure 4.7, Figure 4.8, Figure 4.9), and the results are instead compiled using SPARQL queries.

```
isA(?link,build:DependencyLink) ^
hasDependencyTarget(?link,?repository1) ^
hasDependencySource(?link,?repository2) ^
coveringLicense(?repository1,?license1) ^
coveringLicense(?repository2,?license2) ^
incompatibleWith(?license1,?license2) ^
differentFrom(?license2,?license3)
=> hasSimpleViolation(?repository1,?repository2)
```

Figure 4.7 Type 1 SWRL Rule.

```
isA(?linkA,build:DependencyLink) ^
isA(?linkB,build:DependencyLink) ^
hasDependencyTarget(?linkA,?repository1) ^
hasDependencySource(?linkA,?repository2) ^
hasDependencyTarget(?linkB,?repository2) ^
hasDependencySource(?linkB,?repository3) ^
coveringLicense(?repository1,?license1) ^
coveringLicense(?repository2,?license2) ^
coveringLicense(?repository3,?license3) ^
incompatibleWith(?license1,?license2) ^
differentFrom(?license1,?license2) ^
differentFrom(?license1,?license3) ^
differentFrom(?license2,?license3)
=> hasEmbeddedTransitiveViolation(?repository1,?repository3)
```

Figure 4.8 Type 2 SWRL Rule.

```
isA(?linkA,build:DependencyLink) ^
isA(?linkB,build:DependencyLink) ^
hasDependencyTarget(?linkA,?repository1) ^
hasDependencySource(?linkA,?repository3) ^
hasDependencyTarget(?linkB,?repository2) ^
hasDependencySource(?linkB,?repository3) ^
coveringLicense(?repository1,?license1) ^
coveringLicense(?repository2,?license2) ^
coveringLicense(?repository3,?license3) ^
compatibleWith(?license1,?license3) ^
compatibleWith(?license2,?license3) ^
incompatibleWith(?license1,?license2) ^
differentFrom(?license1,?license2) ^
differentFrom(?license1,?license3) ^
differentFrom(?license2,?license3)
=> hasCompoundTransitiveViolation(?repository1,?repository2)
```

Figure 4.9 Type 3 SWRL Rule.

4.1.5 Global Analysis

Since the previous SPARQL queries and SWRL rules will focus on producing results for specific license violation combinations, further analysis from this combined data is vital to produce a macro perspective that explores trends and compares results. To perform this type of global analysis of the most common license violation pairs and triples (results are discussed in the next section) various queries and small programs were written to derive those violations for each license type. A program was writing for each type of violation. Essentially each program took as input the results of one of the above SPAQRL queries in CSV format. The most frequent license combination was found using a simple for-loop. The counts and combinations were outputted as CSV. This CSV result was then imported into Excel and graphed. These small programs are included as appendices called Type1Analysis.swift, Type2Analysis.swift, Type3Analysis.swift.

Chapter 5

5.1 Results and Evaluation

After defining what new license violations to look for, devising queries for them, amassing the results in CSV format, and finally performing some global analysis, we must now focus on reviewing the found results and evaluate them in critical discussion. Before considering the findings into context, one must recall the primary research statement that states:

> *An approach will be developed based upon a flexible license modelling method that can detect both potential direct and transitive license violations.*

Along with the secondary research statement:

> *Furthermore, we expect that the most common type of violation will be a directly dependent Type 1 Simple Violation, the second most common a transitive Type 2 Embedded, and the third a transitive Type 3 Compound Violation.*

In order to disprove or accept these hypotheses, there should be a substantial number of violations of each type found (Types 1, 2, and 3). Furthermore, if transitive violations of Types 2 and 3 are found, this would indicate that such violations are indeed an area of interest for the research community.

5.1.1 Experimental Setting

The experiments were conducted on an 8-core Intel Core i7-950 clocked at 3.07GHz and 24 gigabytes of RAM. The Linked Data data set is hosted using OpenLink's Virtuoso Universal Server software [60]. The data set is derived from the Apache Maven Project [31]. The data set contains 371 262 projects, and 1 849 756 project releases. Altogether these project releases collectively have 27 934 538 dependencies. Of these, the median is 2 dependencies per project release.

5.1.2 Detected Violations: Trends & Influences

In performing the three queries for detecting the different types of license violations (introduced in Chapter 4 Implementation), our study produced these results:

Table 5.1 Total number of violations reported from the analysis of our data set.

Type 1 Simple Violations	131 996
Type 2 Embedded Violations	288 153
Type 3 Compound Violations	654 964

As the study yielded over 131 000 simple violations and numerous transitive violations of various types, we may conclude that our technique Oswaldo can detect software license violations. It must be noted that these results are only a subset of the unidirectional compatibility that might exist since we only consider 6 of 21 licenses (due to limited availability of compatibility relationships that are vetted by lawyers). Furthermore, because both simple and transitive violations in Open Source Software projects were detected in the Maven repository of software projects, we conclude our primary research statement as confirmed, which will be discussed in further detail below.

From the identified violations, we can first note that Type 3 is seemingly the most common type of violation, followed by Type 2, then 1. This is contrary to our secondary research statement which posited that the most found violations would be of Type 1, followed by Type 2, followed by Type 3. Accordingly, our second research statement is disproved. (Please note that Types 2 and 3 results do not include results from Type 1.) The following three sections investigate found examples of each type of license violation and deliberate their characteristics vis-à-vis the two research statements.

5.1.3 Type 1 Simple Violations

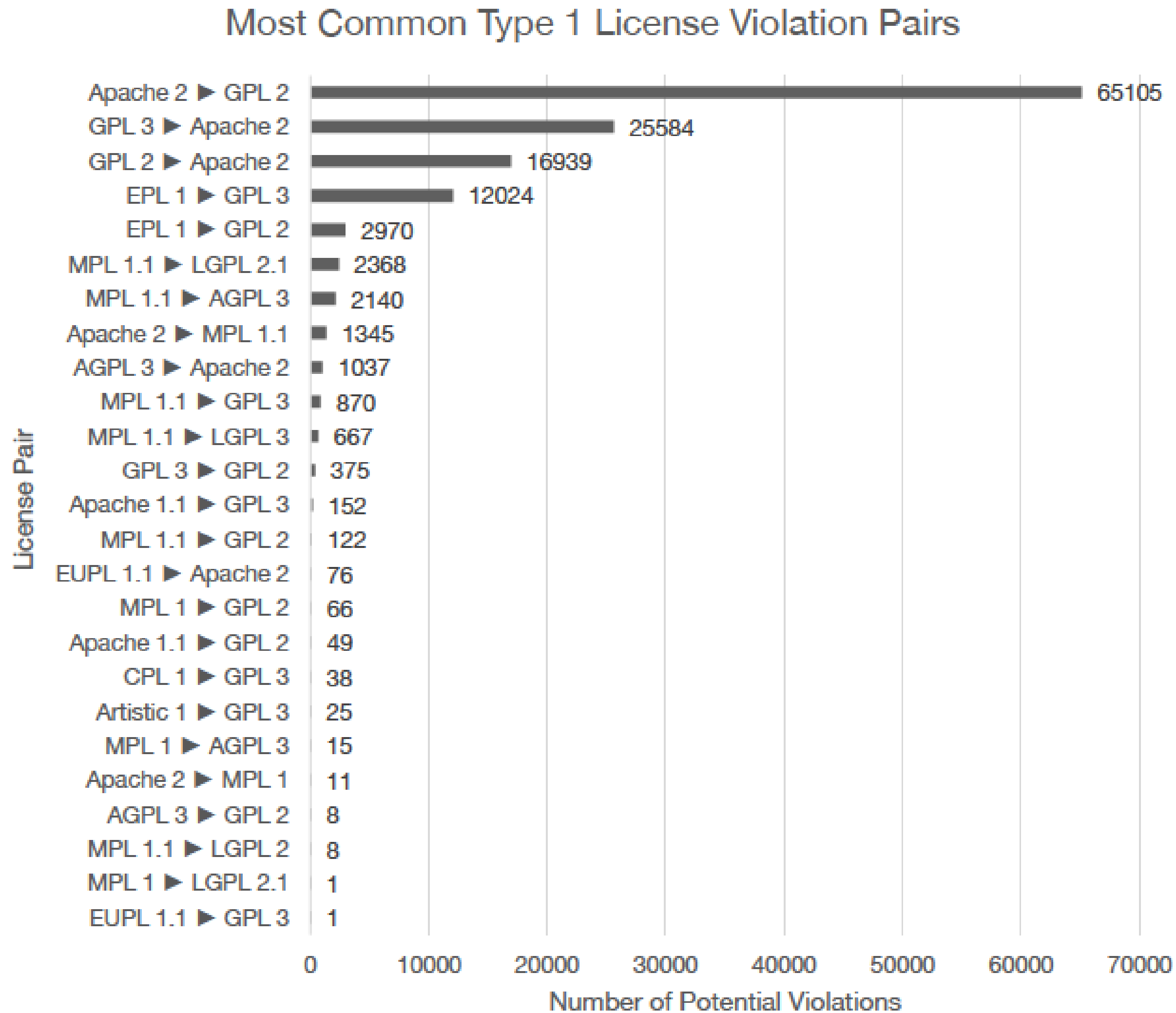


Figure 5.1 Most Numerous License Violation Pairs for Type 1 Simple Violations.

The most common found example for Type 1 Simple Violations in our data set is Apache 2 code being incorporated into GPL 2 licensed code, which represents 49.3% of all Type 1 violations and 6.1% of all three found violation types. This violation is not surprising for two reasons. First, many software developers are simply not aware nor well-versed in open source license compliance, and as these are the two most common licenses in the world, this pairing reflects their usage in the wild. Second, there is likely some confusion about Apache 2's compatibility with the GPL. On the GNU website, the Free Software Foundation publishes a list of licenses that are compatible with

the GPL. This page identified Apache 2 as compatible, but in the license discussion, the authors explain that Apache 2 is only compatible with GPL 3, not GPL 2 [38].

Whereas the first license pair of Apache 2 ▸ GPL 2 could be interpreted as: the programmer attempted to perform due diligence but failed to read the fine print, the second violation pair of GPL 3 ▸ Apache 2 could be seen as more blatant. The GPL 3 is widely known to be a more restrictive license. The programmer who committed this type of violation 1) must have either wilfully broken copyright law, or 2) must have been unaware that open source licenses differ vastly in their allowances. First, if the developer knew the terms of the GPL 3 license and chose to import the project into the Apache 2 project as is, and not change the license of her downstream project (according to the reciprocity clause in the GPL 3's terms and conditions) this would amount to an intentional violation of copyright law, thus the programmer (or the organization) would be held liable if sued (and if that intentionality could be proven, would most likely result in a stronger sentence). Second, if the developer was unaware of how OSS licenses work, they could still be taken to court. If taking the latter more charitable vantage point, one would conclude that (despite the compatibility listing [38]) the developer was ignorant of the fact that license compatibility is unidirectional. Thus, this assumption induced a violation of the GPL 3 because that project was imported into an Apache 2 project.

When comparing the number of found Type 1 violations to Types 2 and 3, we can observe that Types 2 and 3 are more frequent. This may be because a Type 1 Simple Violations is a better-known compatibility/incompatibility. That is, it is much more likely to be discovered by developers, since it only involves two licenses, and is a straightforward relationship. The transitive types on the other hand, have not been considered in the research community before this thesis, and some apparent violations may very well be acceptable (depending on the mix of conditions) and not violations at all.

For example, the European Union Public License (EUPL) explicitly states which licenses it is compatible with. This is a known compatibility. Whereas for transitive interactions, the EUPL may then be imported into an intermediary project, say a project under the Licence Libre du Québec–Réciprocité (LiLiQ-R), which is then imported into a tertiary project under Common Development

and Distribution License (CDDL). Each step (EUPL to LiLiQ-R, and LiLiQ-R to CDDL) are known to be compatible. Even though the direct relation of EPUL to CDDL is not compatible, the use of an intermediary license may enable compatibility. This chain of licenses would be flagged as a violation by Oswaldo, although it could in fact be lawful (verifiable by a lawyer). The existence of such ambiguity would then contribute to the number of Type 2 violations found.

5.1.4 Type 2 Embedded Violations

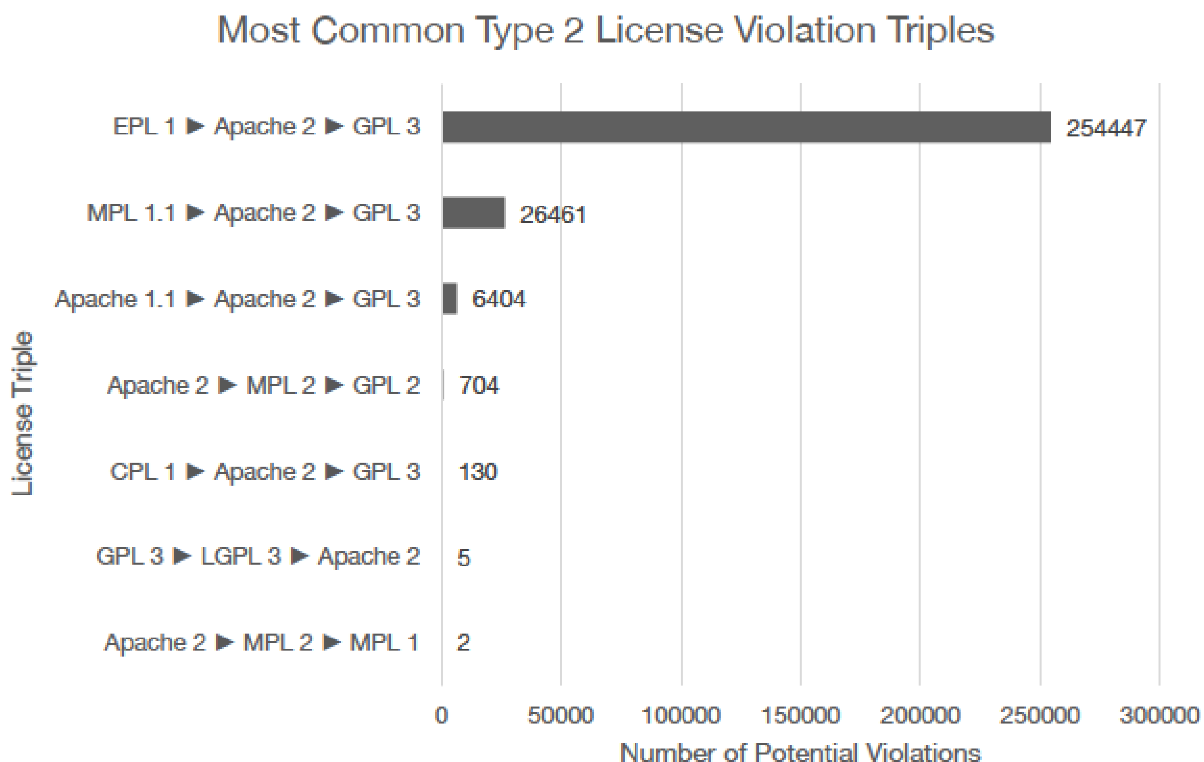


Figure 5.2 Most Numerous License Violation Pairs for Type 2 Embedded Violations.

For Type 2 Embedded Violations, the most prevalent triple of licenses is EPL 1 ▸ Apache 2 ▸ GPL 3, where EPL 1 is incompatible with GPL 3, which represents 24.3% of all Type 2 relations. This incompatibility is particularly tough for a human to spot because the EPL 1 is importable into Apache 2, and Apache 2 is importable into GPL 3. It would require a manual analysis of the Apache 2 project to discover the embedded EPL 1 dependency, which is incompatible with the GPL 3, as stated by the FSF's legal team in their compatibility list [38].

MPL 1.1 ▸ Apache 2 ▸ GPL 3 is the second most common Type 2 dependency triple, where the MPL 1.1 cannot be imported into the GPL 3. The FSF states: "Software under previous versions of the MPL can be upgraded to version 2.0, but … software that's only available under previous versions of the MPL is still incompatible with the GPL [38]." Consequently, it would be theoretically conceivable to resolve the 26 461 found violations by upgrading the MPL from version 1.1 to version 2. In practice, however, this migration process can be cumbersome; obtaining consent concerning the license upgrade may be required from all project authors (i.e. anyone who made contributions), which could be in the thousands for a popular project. "Without copyright assignment or a CLAs [Contributor License Agreements], changing a software license requires the consent of every contributor to that system [61]." For many projects, this license evolution process is infeasible, which then returns us to the initial unresolved Type 2 conflict between MPL 1.1 and GPL 3. This conflict continues to stand as an actual violation.

Finding these well-known Type 2 violations with Oswaldo leads to another question; why have so many Type 2 violations not been found by project authors? We can theorize why. First, it is difficult and extensively time-consuming for a programmer to trace import statements in the source code to reveal the exact dependency hierarchy of a project. This nuisance is one of the reasons why no detailed license inspection for these transitive dependencies is performed. Second, as FLOSS licenses have many terms and conditions, programmers may be lacking 'FLOSS literacy.' That is, they may not know enough of the basics to properly use licenses. Third, there does not exist a centralized, concise, and well-defined set of rules or guidelines for the use of and interaction among all major licenses. This makes it difficult for a developer to not only to select an appropriate license but also to aid the detection of license violations by the developer.

5.1.5 Type 3 Compound Violations

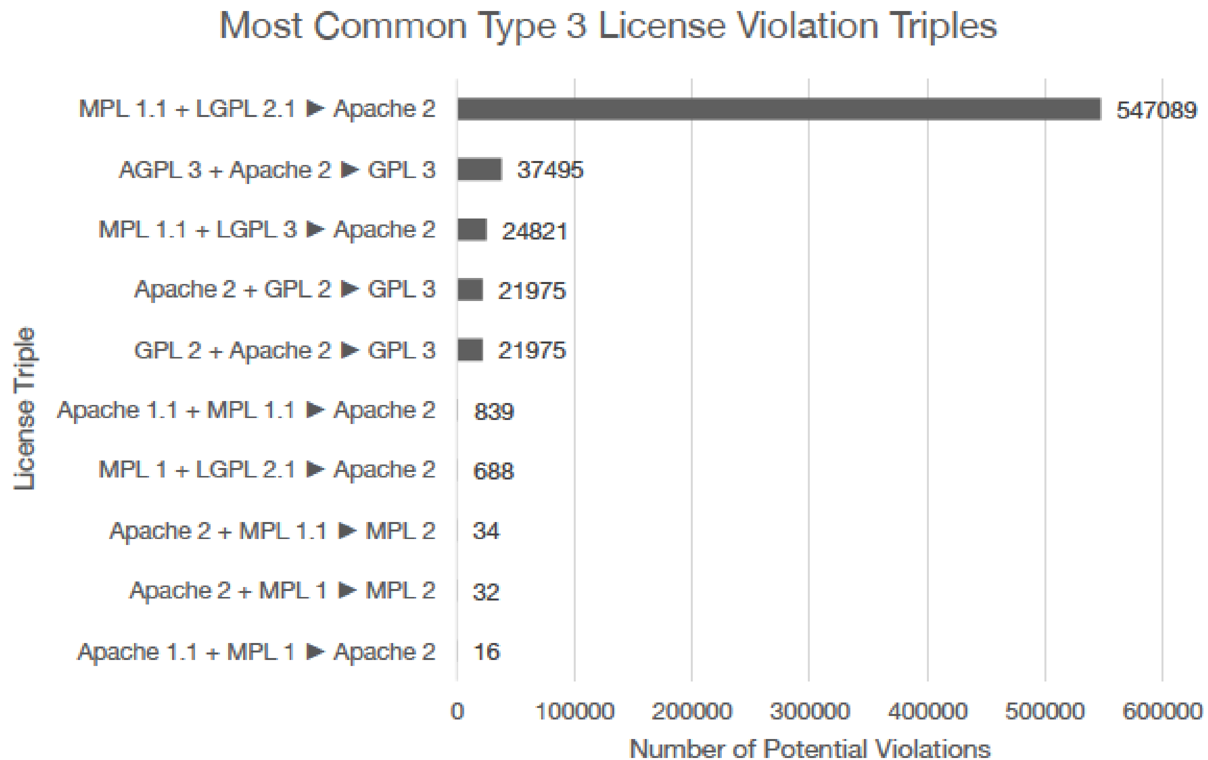


Figure 5.3 Most Numerous License Violation Pairs for Type 3 Compound Violations.

After executing the Type 3 Compound Violation query, the most widespread license conflict that we observed in our data set was MPL 1.1 + LGPL 2.1 ▸ Apache 2 (shown below in Figure 5.4), which represents 52.2% of all Type 3 violation relations. Within this use case, MPL 1.1 is discordant with LGPL 2.1. However, some of these triples may not be a violation. If the LGPL 2.1 project is only linked to then according to the terms of the LGPL, this is not a violation.

As previously discussed when clarifying the meaning of compatibility, according to the LGPL linking stands for "a work that uses the library." This usage can be static (compile-time) or dynamic (run-time) as long as both projects' *source* code is kept separate. Each project is an "independent work that stands by itself, and includes no source code from [the other]." It is perfectly acceptable to combine *compiled* code however [56].

Moreover, keeping this definition in mind, linking is by far the most likely use for the LGPL since this license was specially created for linking libraries [57]. If the project was not linked, but source-code-combined, then this license triple combination would be a definitive violation. However, the downstream project is Apache 2, which is a weaker copyleft license. This makes it highly likely that the LGPL project is being linked (and not combined with) the Apache project. Thus, it is recommended that the reader holds this caveat in mind while moving onto the next Type 3 example.

Since we have considered the definition of "linking" by the LGPL, we should also consider the definition of linking by the MPL as well. The MPL allows for static linking unlike the LGPL ("Distribution of a Larger Work" [62] [63]). This mismatch of definitions could lead to much confusion when integrating these projects and their licenses. If the MPL project were linked directly to the LGPL project a violation could occur (Type 1). But this is not the case as the linking occurs inside the Apache 2 project (Type 3). However, the fact still stands that the LGPL is still most likely dynamically linked to the Apache 2 project, and therefore this is likely not a violation.

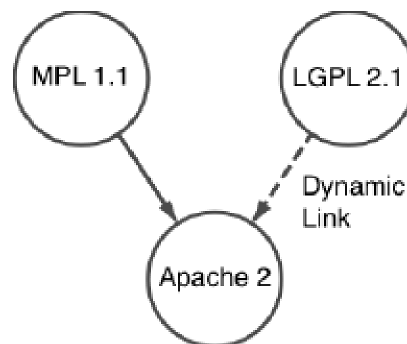


Figure 5.4 Type 3 Relationship with Dynamic Link.

The second most found license triplet with 37 495 results, is AGPL 3 + Apache 2 ▸ GPL 3. Again, the incongruity is between the two sibling licenses in the dependency hierarchy: AGPL 3 cannot be imported into Apache 2. However, Apache 2 can be pulled into AGPL 3. Because these licenses are siblings, and the Apache 2 code will be relicensed as GPL 3 code, it is conclusive that these two projects can comingle together without any problems.

Looking at the third found triple of MPL 1.1 + LGPL 3 ▸ Apache 2 licence dependencies, this license hierarchy is similar to the first Type 3 example discussed above.

Let us continue to the fourth and fifth Type 3 examples: Apache 2 + GPL 2 ▸ GPL 3. Apache 2 code is *not* importable into GPL 2, and vice versa. Again, a conflict may likely not occur between these two licenses due to their transformation into GPL3 code.

Considering the subsequent triple of Apache 1.1 + MPL 1.1 ▸ Apache 2, the two dependencies are incompatible, where Apache 1.1 code cannot be used in MPL 1.1 code. However, this particular triple is slightly and subtly different. In this case, a violation could possibly occur because the MPL's definition of linking is different. The MPL allows static linking of source code, i.e. the file can be placed in the project and built. (This placement is allowed because the license can be applied on a file-level, as opposed to a project-level. See section "3.7. Larger Works" of the MPL 1.1 [62].) However, Oswaldo does not model pure file-based relationships; the project relationships are derived from Apache Maven's POM (Project Object Model) files, and these POM-file-derived relationships may not take into account manually imported and statically linked MPL files. Thus, a violation could very well occur if the programmer has not paid attention to the combination of all three of these licenses' terms. Therefore, Oswaldo currently cannot distinguish if a reported Type 3 result is an actual violation.

Below we consider the remaining license triads that include potential violations:

- MPL 1 + LGPL 2.1 ▸ Apache 2
- Apache 2 + MPL 1.1 ▸ MPL 2
- Apache 2 + MPL 1 ▸ MPL 2
- Apache 1.1 + MPL 1 ▸ Apache 2

These license triples all succumb to the complexities described above, mainly the mixing of differing definitions of linking and the modelling shortcomings of the Apache Maven's POM files.

One more example of importance must be discussed. There exist modified versions of the GPL 2 license which include a reciprocity clause where any modifications or derivative works must be licensed under the GPL 2 only (and not any future versions of the GPL). This is important because it means that that version of the GPL 2 is not compatible with the GPL 3. One prominent example of this is Linux itself, where the original author of the operating system, Linus Torvalds, refuses to upgrade to GPL 3 [64]. (Linux's license is the GPL 2 only version.) Combining two 'parental' dependencies where one project is GPL 2 only and the other project is GPL 3 would create a license violation in the child project. Keep in mind that the Oswaldo license model currently cannot distinguish between this subtle difference. Oswaldo simply considers all GPL 2 licenses to be the same. Modelling this subtlety would likely increase the quality and number of Type 3 violations found.

5.1.5.1 Type 3 Example

Even though we did not find a clear Type 3 violation using Oswaldo, three licenses used together can result in a real violation. For example, a developer uses three projects together in the relationship shown below in Figure 5.5. The currently developed project is BSD-licensed and has two dependencies: one project that is licensed under Apache 2, and another Artistic 1.

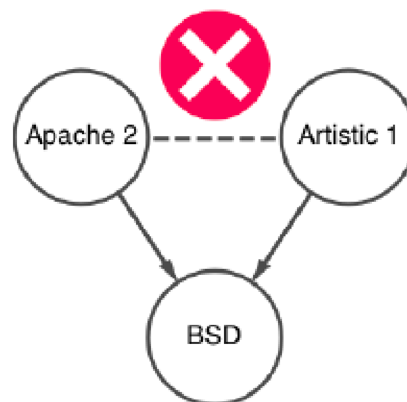


Figure 5.5 Type 3 Violation occurring between the Apache 2 project and Artistic 1 project, because both are used together in the BSD project.

In this usage scenario, the violation occurs between the two dependencies. Artistic 1 cannot be used within Apache 2 code. When the BSD project is compiled and run, so are its dependencies

which would then mean the resultant binary contains code from all three projects. Thus a Type 3 violation would occur.

In conclusion, even though the evidence for Type 3 violations in this study is might seem unconvincing, there remains a convincing argument and strong possibility that such violations do occur. The situation is more complicated than what is currently modelled in our ontology. Presently, Oswaldo only models six out of twenty-one most-popular open source licenses, and thus may be missing pertinent incompatible license relationships. This expansion of the ontology and further investigation is left to future research.

5.1.6 Evaluating Actual (and Notional) License Violations

In order to further explore our primary research statement, we must investigate if the found violations are indeed actual violations. We conduct this investigation while keeping in mind that ultimately the final decision of any violation discussed here will be left to lawyers specializing in copyright violations and the courts. There are multiple facets to an actual violation result. The first is the definition of a violation. And the second facet is the structure of the violation itself.

Creating a definition of a violation is not a simple task either. Each license may define exactly what this term means in its context. Thus, one must investigate various licenses and their detailed terms and conditions. Hereto (with respect to bundling and importing code between heterogeneously-licensed projects) we explore what constitutes a violation in the GPL, LGPL, and MPL.

For the GPL, the book *Understanding Open Source and Free Software Licensing* delves into the specifics of a violation: “Accordingly, if the other program were licensed under a proprietary license and the library under the GPL and the program and library were distributed together under the proprietary license, the GPL would be violated, as the program plus library would be considered a derivative work that would be subject to limitations on copying, distribution, and modification that are inconsistent with the GPL.” Furthermore, “the use of a GPL-licensed program with a proprietary-licensed library (or any other program, whether under a proprietary license or some other non-GPL license) is not a violation of the GPL license. Rather, the GPL

license comes into play only when the GPL-licensed software is copied, distributed, or modified—none of which is implicated by the simple use of the software [17]." Essentially the question of committing a violation boils down to whether a derivative work is created or not, when combining dependencies into a new project.

The LGPL states: "the object code form of **an Application may incorporate material from a header file that is part of the Library**. You may convey such object code under terms of your choice, provided that, … you do both of the following: a) Give prominent notice with each copy of the object … b) Accompany the object code with a copy of the GNU GPL and this license document [65]." Put simply, in order to comply with the LGPL, the software developer needs to make a bundle of two things: the project binary and the source code from the LGPL library (which includes the LGPL license file). This is a fairly easy requirement to spot and fulfill when one has a Type 1 Simple dependency. However, when one has to deal with a Type 2 Embedded dependency this task becomes inherently more difficult. For example, the intermediary dependency may not require the propagation of source code, but the top-most dependency may require propagation. Since one inherently uses the top-most dependency through the use of the intermediary dependency, one would cause a violation if one does not end up distributing the source code to the top-most library. Since the decision was taken to mark the LGPL as compatible with various licenses in Oswaldo (as described previously), the number of found violations is likely underrepresented. This supposition is due to the fact that linking information (i.e. statically or dynamically linked dependencies) are currently not modelled.

The MPL has a subtly different definition of a violation, because the MPL has different requirements for non-MPL-licensed juxtaposed code. The MPL 2 states in Section 3.3 Distribution of a Larger Work: "If the Larger Work is a combination of [MPL-]Covered Software with a work governed by one or more Secondary Licenses, and the Covered Software is not Incompatible With Secondary Licenses, this License permits You to additionally distribute such Covered Software under the terms of such Secondary License(s), so that the recipient of the Larger Work may, at their option, further distribute the Covered Software under the terms of either this License or such Secondary License(s) [63]." Therefore, one is allowed to mix (copy and paste) an MPL-licensed file into a project under another license (provided both licenses' terms can be satisfied

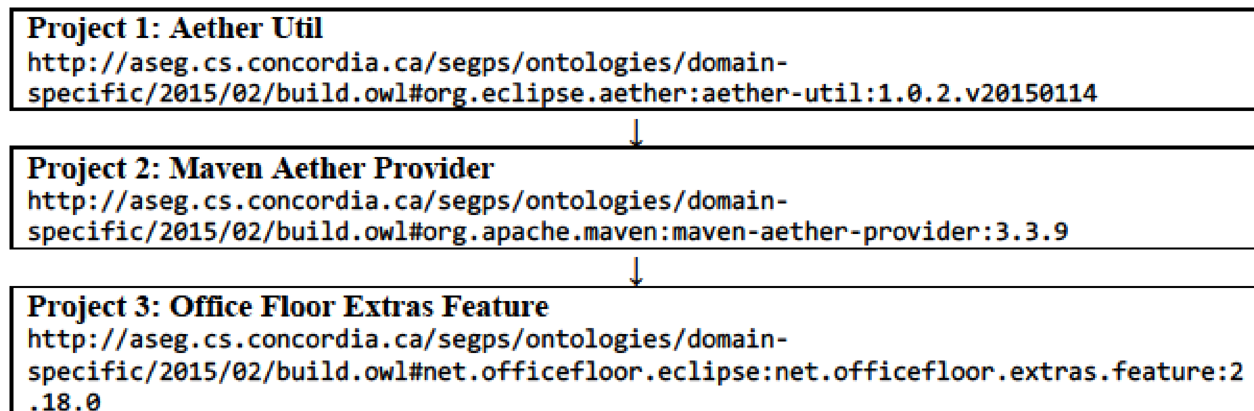
simultaneously). The resulting binary of this work can then be distributed under either license. Thus, a violation would only occur if: 1. The other license was known to be incompatible, 2. Some condition of the MPL was not followed i.e. the copyright notice was removed from the MPL-licensed file. The key difference to bear in mind here is that MPL violations deriving from multi-licensed projects are file-based unlike the GPL and LGPL that are project-based.

Now that the detailed terms and conditions of licenses have been examined to determine the veracity of a violation, let us conclude that all three licenses (GPL, LGPL, MPL) clearly show that they can be violated when improperly combining projects together. Even though many licenses have detailed conditions for multi-project use cases, the ultimate judgement on whether an actual violation has occurred, is the decision of a lawyer or judge.

5.1.6.1 Notional Example

Next, we explore the detailed structure of the projects which make up a transitive violation by meticulously cloning each project repository (and each of its dependencies) to determine if indeed an actual violation has occurred. Specifically, the way the linking of projects is structured. We start off with a found Type 2 example of EPL 1 ▸ Apache 2 ▸ GPL 3.

Oswaldo found the following potential violation:



Project 1: Aether Util
http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#org.eclipse.aether:aether-util:1.0.2.v20150114

↓

Project 2: Maven Aether Provider
http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#org.apache.maven:maven-aether-provider:3.3.9

↓

Project 3: Office Floor Extras Feature
http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#net.officefloor.eclipse:net.officefloor.extras.feature:2.18.0

A manual inspection of the Extras Feature project did not reveal any import statements in the Java code that linked to the Maven Aether Provider project. Furthermore, many of the other Office

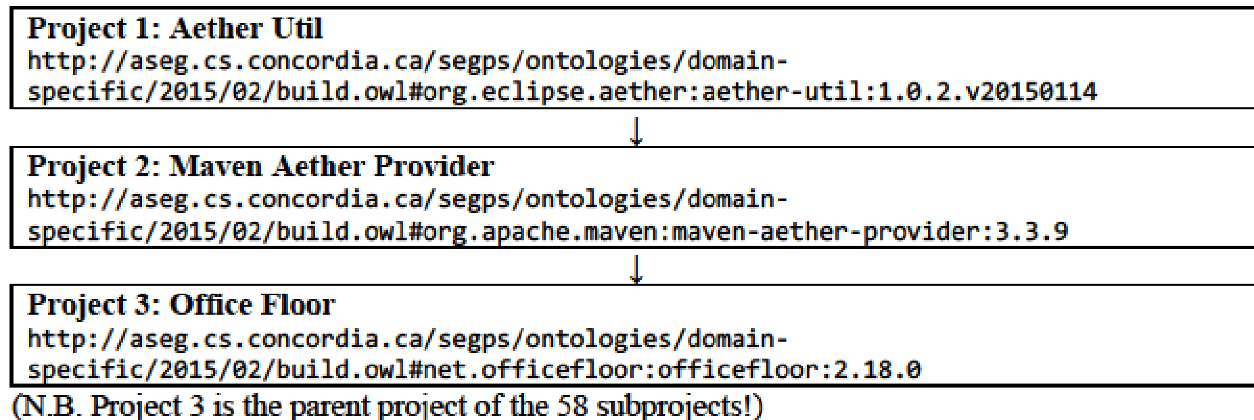
Floor projects (58 other projects in total) were reported as potential violations. A subset of such projects were:

- net.officefloor.core:officeframe
- net.officefloor.ui
- net.officefloor.woof.feature
- net.officefloor.plugin:officeplugin_base
- net.officefloor.core:officebuilding

After extensive additional manual investigation of multiple Java files in multiple projects which resulted in no concrete link found, the veracity of the POM file came into question. In fact, scrutinizing the POM files showed that they inherit dependencies from their parent POM files. To conclude, all 58 of these Office Floor subprojects have to therefore be excluded from the results.

5.1.6.2 Actual Example

Alongside the 58 subprojects, Oswaldo found one more potential violation of Type 2: EPL 1 ▸ Apache 2 ▸ GPL 3, where the first project cannot be used in the third project.



(N.B. Project 3 is the parent project of the 58 subprojects!)

In order to validate that all three projects are linked through import statements, we must:

- identify the use of project 2 (from project 3),
- identify the use of project 1 (from project 2).

We began by scrutinizing the Office Floor project. Our analysis showed that the file `ClassPathFactoryImpl.java` which imports the file `MavenRepositorySystemUtils` on line 33.

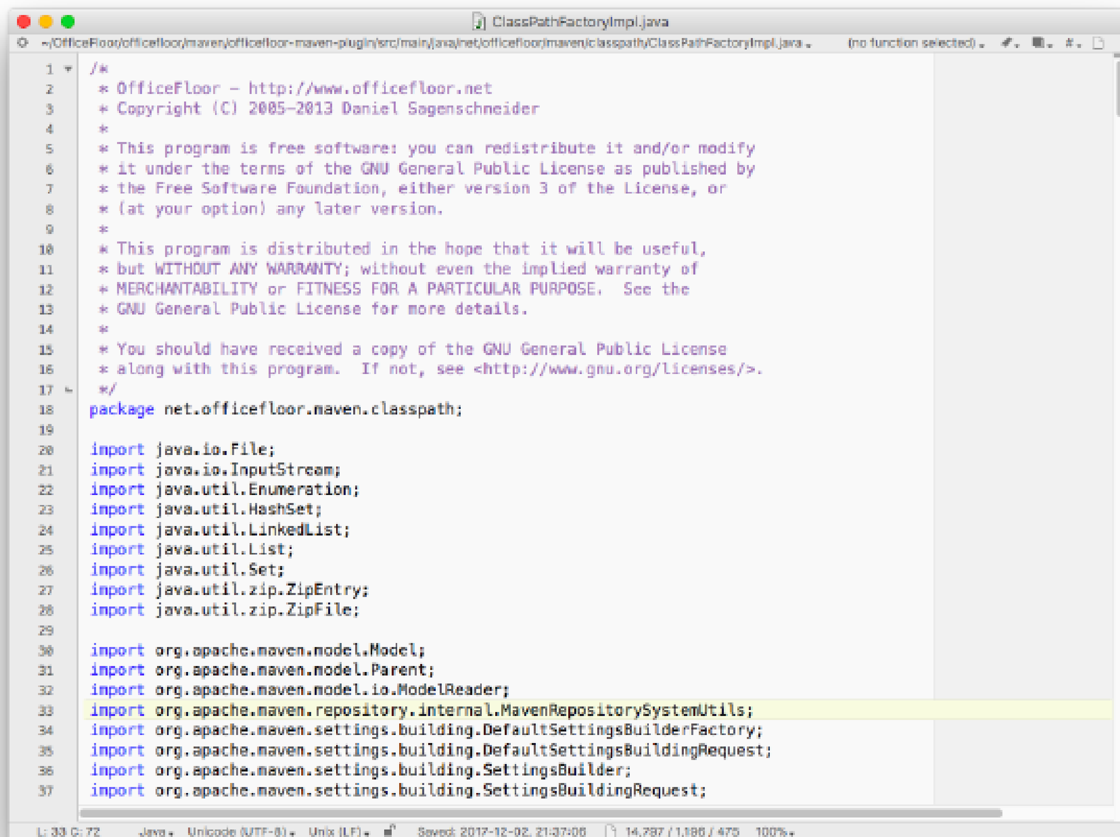


Figure 5.6 ClassPathFactoryImpl.java of the project Office Floor, version 2.18.0.

As part of the analysis, a grep search was performed to locate the import file in the Maven Aether Provider project (statement #33, highlighted in Figure 5.6).

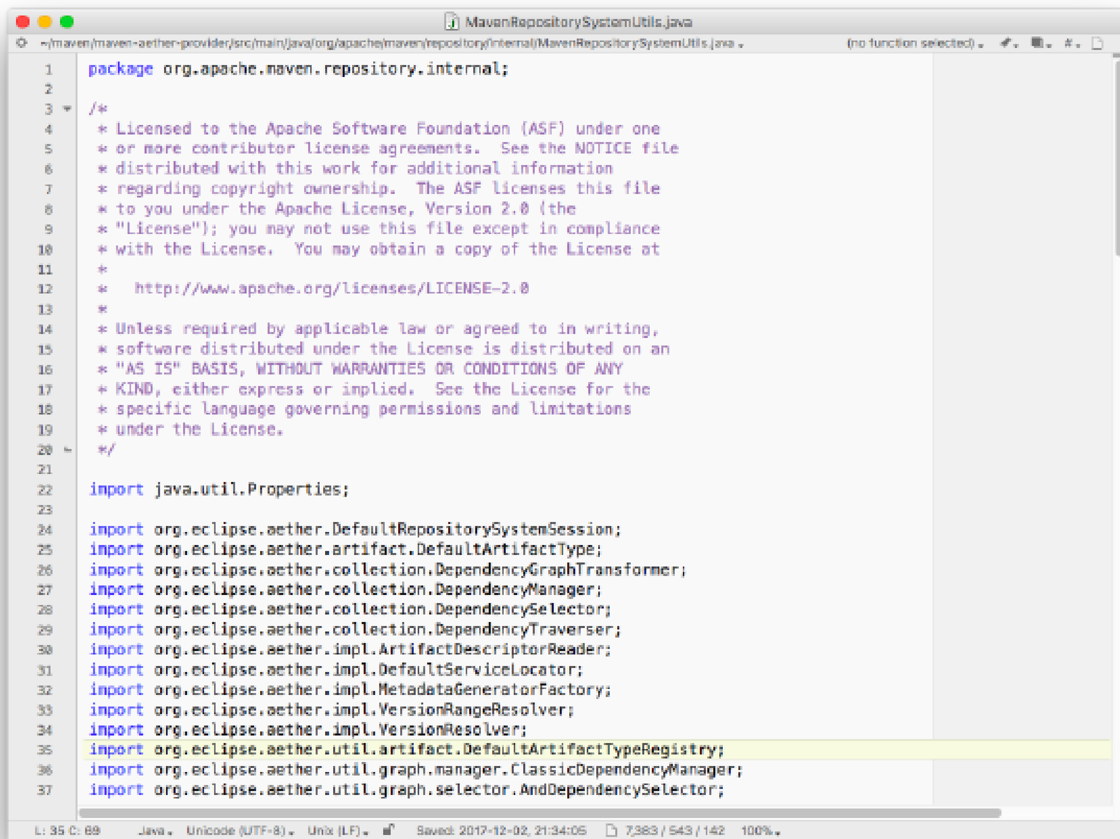


Figure 5.7 MavenRepositorySystemUtils.java of the project Maven Aether Provider, version 3.3.9.

Continuing the investigation through the dependency hierarchy to verify the license violation, we search in the just found `MavenRepositorySystemUtils.java` file (Figure 5.7) of the Maven Aether Provider project and identify the import statement on line 35 pointing to `DefaultArtifactTypeRegistry` of the Aether Util project (Figure 5.8).

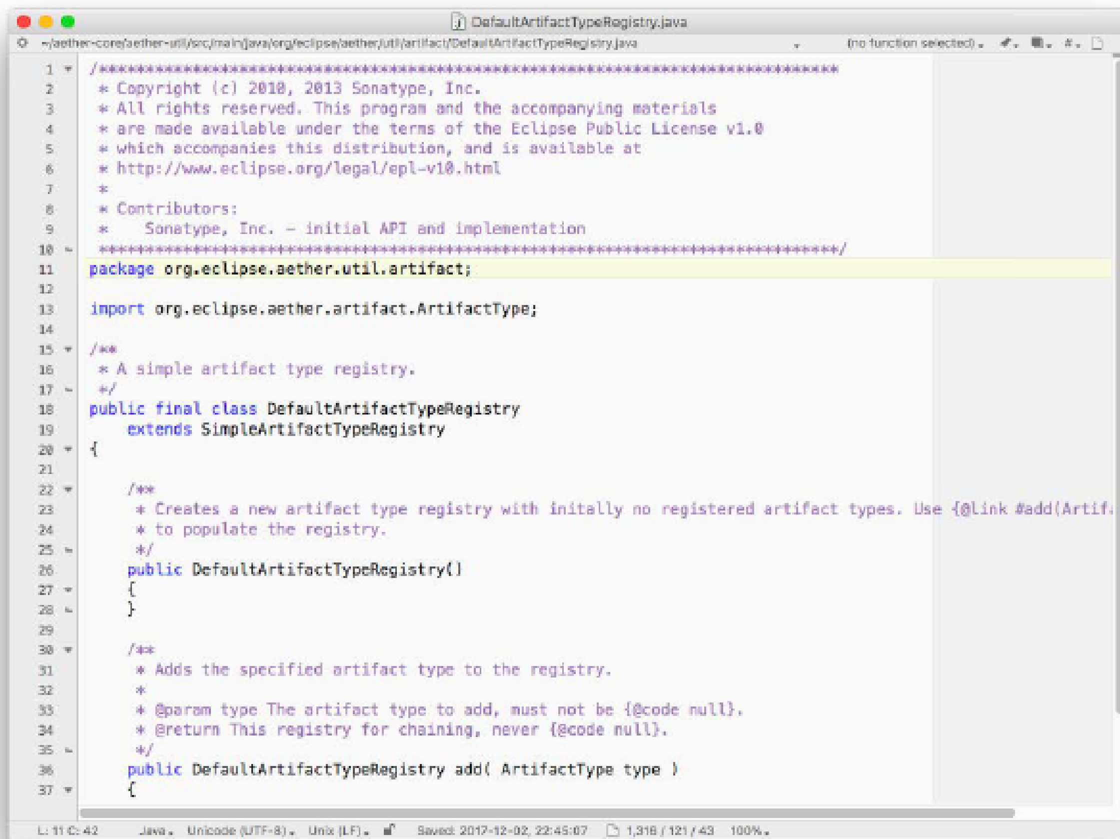


Figure 5.8 DefaultArtifactTypeRegistry.java of the project Aether Util, version 1.0.2.v20150114.

We have shown that all three projects are indeed used together as found by Oswaldo. A Type 2 violation occurs because of an incompatibility between the top-most (Project 1) and the bottom-most project (Project 3), through an intermediary project (Project 2). In this violation's case, we have the EPL 1 that is irreconcilable with the GPL 3. On the GPL-compatible license list webpage, the EPL entry states: "The Eclipse Public License is similar to the Common Public License, and our comments on the CPL apply equally to the EPL [38]." The CPL entry says unequivocally: "This is a free software license. Unfortunately, its weak copyleft and choice of law clause make it incompatible with the GNU GPL [38]." That clearly states that the EPL 1 is not compatible with the GPL 3. Thus, we can conclude that this is a confirmed violation surfaced by Oswaldo.

5.1.6.3 Frequency of Violation Types Reconsidered

The following table is presented again with updated totals, taking into account some of the complexities discussed above. As each of the found license combinations were examined with some combinations suspected to likely not be license violations, these were subtracted from the totals as noted in the following paragraphs.

Table 5.2 Total number of violations revisited.

	Original	Revised
Type 1 Simple Violations	131 996	131 996
Type 2 Embedded Violations	288 153	281 744
Type 3 Compound Violations	654 964	921

As our definition of compatible already makes an exception for the LGPL, the number of Type 1 Simple Violations do not need to be adjusted.

The amount of Type 2 violations was amended to specifically exclude the triple Apache 1.1 ▸ Apache 2 ▸ GPL 3 because this is a likely a case of the Apache licence being upgraded to version 2 and thus removes the conflict. A second triple was removed of GPL 3 ▸ LGPL 3 ▸ Apache 2 due to the likelihood of being dynamically linked.

Type 3 violations were retallied by excluding any triples with any version of the LGPL (because the primary use-case is meant for dynamic linking) and any triples where a weak copyleft dependency is relicensed under a strong copyleft that then makes the code compatible. To be clear, the following triples were kept:

- Apache 1.1 + MPL 1.1 ▸ Apache 2
- Apache 2 + MPL 1.1 ▸ MPL 2
- Apache 2 + MPL 1 ▸ MPL 2
- Apache 1.1 + MPL 1 ▸ Apache 2

Now we can revisit the secondary research statement is:

> *Furthermore, we expect that the most common type of violation will be a directly dependent Type 1 Simple Violation, the second most common a transitive Type 2 Embedded, and the third a transitive Type 3 Compound Violation.*

Accordingly, we conclude that the secondary research statement is false. Indeed, Type 2 Embedded Violations are the most common, followed by Type 1 Simple Violations, followed by Type 3 Compound Violations.

5.1.7 Findings

Based on the three license-specific definitions and project structures discussed and explored previously, do transitive violations actually exist? Yes, our evaluation of Maven clearly shows that transitive violations do exist.

We observed a large number of Type 1 Simple Violations (over 130k). Given that even the most straightforward relationship type (a direct one-to-one relationship) resulted in license violations, this may indicate that there exists a large knowledge gap of — or worse, a blatant disregard for — FLOSS licenses and their inter-usage. Also of note, is that even though the total number of Type 1 Simple Violations is less than the number of transitive violations, there is more difference and variety in the licenses involved in a violation. This may be because using two licenses together is more common than using three together. It may also be due to the fact that Oswaldo currently models only a subset (6 of 21 licenses) of unidirectional compatibility.

An even larger amount of Type 2 Embedded Violations was found (over 281k). The fact that there is an even larger number, and this transitive violation type has never been considered before, is intriguing. Transitive violations are indeed a problem in the software development community, overlooked until now. This is most likely due to the time-intensive process to check all dependencies (and all of the dependencies' dependencies) through import statements or package managers for compatibility between various licenses. And since having never been considered

before, there is less of a user awareness of what constitutes transitive violations, making it difficult to check for and resolve a problem that one did not know existed.

The results for Type 3 Compound Violations were more intricate, and thus many results were excluded resulting in only 921 triples. Further complicating the situation of Type 3 violations is that for the modelled licenses, without the direct input of a copyright lawyer, these rules can just not be defined well enough with respect to differing linking definitions between various licenses as well as subsumption and relicensing clauses. However, a concrete triple was thoroughly explored which shows that a Type 3 violation can indeed exist.

5.1.8 Threats to Validity

In what follows, we discuss some potential threats to validity for our research and the experiments conducted.

We have searched extensively and made strenuous effort to ensure the "compatibleWith" relationship in the ontology is correct, however this is not legal advice and any liability of the reuse of these relationships rests upon the developer reusing them. In future, this research would benefit from a review and approval from multiple legal counsel familiar with copyright laws worldwide.

Only a very limited number of license violations are currently modelled in Oswaldo's ontology. By extending the ontology model, the chances to potentially identify extended violations will increase. In order to find all possible violations, 882 compatibility relationships would need to be defined since there are 21 licenses (including each license version). That is, there is a 21 by 21 matrix, equalling 441 relationships. Yet compatibility is not bidirectional but unidirectional. Thus, the amount is doubled to 882. Still, this number could be much larger if more than the 21 major licenses (and license versions) were added to the ontology.

We assume the license information from Maven to be accurate (which, as shown with the notional example, is not always the case). A POM file assumes that all files in one project are licensed the same, thus the granularity of the license information is project-level. In fact, some variance in style occurs on how developers mark that a project is released under a certain license. Sometimes the

developer puts a "LICENSE.txt" file in the root directory of the repository, thus all the files in the repo are supposedly released under that one license. However, another style exists where the license is copied and pasted at the top of each source file. This can be one example of a project having multiple licenses. We are assuming Maven's data is correct. We also do not address the complications of projects that are dual- or tri-licensed. For example, Mozilla tri-licensed the popular Firefox web browser (under MPL 1.1, LGPL 2.1, GPL 2) for a number of years to make the project compatible with the GPL and LGPL, for the benefit of GNU and Linux users before they created the MPL version 2 (which is now compatible with the GPL license family) [66], [67].

5.1.9 Evaluation Summary

As our evaluation shows, Oswaldo can detect many violations that were previously not considered by the research community (Type 2 and Type 3 violations). Violation types were defined and searched for in the Maven data set which contains hundreds of thousands of projects (371 262 total). Two of the three types of violations were found and expanded upon. The third type was expanded upon on as well. The first directly dependent type (where one project is imported into another) was defined as a Type 1 Simple Violation. 131 996 of these license violations were found. The second transitive type is a Type 2 Embedded Violation, where ultimately 281 744 of these violations were observed by Oswaldo. In evaluating some of the Type 2 results were discarded due to inconsistencies in Maven's POM.xml files (Project Object Model). We ascertained a project's POM file would inherit the dependencies of its parent POM file. (Which we later showed, the parent project did contain an actual violation.) The third type of license violation was classified as a Type 3 Compound Violation. In the end, there were 921 Type 3 violations, after setting aside results due to critical evaluation including: differences in how various licenses permit static and dynamic linking, and the ontology currently models compatibility relationships between six of twenty-one most-popular open source licenses due to lack of verifiable information from legal professionals. Overall, both direct and transitive violations were shown to be a problem in current open source projects.

Chapter 6

6.1 Related Work

First, we briefly review code clones as they are the finest granularity of code reuse. Finally, we move onto currently defined license violations and detection techniques.

6.1.1 Code Clone Detection

Using code clones to detect small-scale license violations were touched upon by Monden et al. [68] Three quality metrics for code clone detection were compared and contrasted: 1. MLC: maximum length of clones, 2. NCP: number of clone pairs, and 3. LSim: clone-based local similarity ("the percentage of duplication within a suspicious pair"). MLC and LSim were found to be the most effect measures for judging the quality of clones found by the existing CCFinderX tool. These two measures were then combined and evaluated for an even higher level of accuracy. Disappointingly, the authors did not find any actual license violations in OSS. License violations were merely used as a theoretical use case for their comparison study.

The Binary Analysis Tool (BAT) developed by Hemel et al. [10] employs three different techniques to detect code clones of OSS in proprietary binaries for the express purpose of finding violations of popular GPL projects. The authors used the comparison of string literals, data compression, and binary deltas. **String literals** are searched for using GNU `strings` tool, and then they are ranked based on likelihood of being a code clone. **Data compression** was used to reveal code clones by the following logic. Both the proprietary binary in question and the precompiled OSS binary are concatenated together and compressed into one file. At the same time, the size of the compressed proprietary binary plus the size of the compressed open source binary (two separate files, as opposed to one file) is calculated. If the total size of the first combined compressed file is substantially smaller than the size of the summation of the second separate files, then this is a likely indication of redundancy, and thus code cloning. In a similar manner to data compression, the size of a **binary delta** is computed to determine code clones. A delta (also known as a diff or patch) is made from the proprietary binary to the open source binary. If the delta is substantially

smaller than the open source binary, then this is likely evidence of a code clone. There are various limitations to all three approaches, such as false positives of string literal matches from other packages, small binaries are not suitable for data compression, binary delta and data compression methods are architecture dependent, and require precompiled open source binaries, etc. Interestingly BAT does find many true code clones, but falls short by leaving the verification as a manual process, i.e. whether a code clone is also a license violation.

6.1.2 License Violation Detection Tools and Approaches

Di Penta, German, Guéhéneuc, and Antoniol conducted a study on the evolution of software licenses [69]. Six open source projects were focused on: ArgoUML, Eclipse-JDT, FreeBSD kernel, Mozilla Suite, OpenBSD kernel, and Samba. The authors also proposed "an approach to automatically track the licensing evolution of systems, identifying changes in licenses and copyright years." The method consisted of 1. extracting the licensing statements from the comments at the top of each file, 2. using the diffs from the version control system to find when those lines changed, 3. using the license detection tool FOSSology [7] to classify the license, and 4. extracting and identifying any changes in the copyright years. They found that OSS projects do change licenses over time and these changes were not just to new versions of the existing license. i.e. "from one license to another, license additions, e.g., files without license were updated with a license, and license modifications." Sometimes projects who switched licenses altogether had intended and unintended effects on downstream users of these projects.

In relation to this dissertation, seminal research was conducted in 2010 and entitled, "License Integration Patterns: Addressing License Mismatches in Component-Based Development" [22]. The use of two or more software components with differing license terms was explored by Daniel German and Ahmed Hassan at the University of Victoria and Queen's University. They introduced this concept as the "license-mismatch problem" which is analogous to this dissertation's terminology of *license incompatibility or violation*. The authors created a "model to describe licenses, and the implications of licenses on the reuse of components." This model describes what usage scenarios result in a derived work or not, those being: 1. Linking, 2. Fork, 3. Subclass, 4. Intercommunication Protocol 5. Plugin. Interestingly, the authors briefly model the open source licenses using a graph: each software application is a node, and each component (dependency) is

a node, with edges linking the two. Twelve patterns to resolve or circumvent the problem of license incompatibility were described for the licensor and licensee as outlined in Figure 6.1.

Type	Name	Intent
Licensor	Exception	To allow a particular use by expanding the terms of the license in an addendum, without modifying the text of the license itself.
	Disjunctive	To give the option to the licensor to choose one of several licenses that will best suit her purpose.
	Clarification	Give an interpretation of contentious or ambiguous parts of the license.
	Permit Relicensing	Allow the derivative work to be licensed under a different license than the one under which the component is made available.
	Add-on	Allow components under a non-compatible license to extend the functionality of another component via a well-defined API.
	Indirect License	A component indicates that its license will be the same as another one.
	Different parts, different licenses	Provide different parts or features of the system under different licenses.
Licensee	Patch	Issue a patch that the user can apply to the component to create the derivative work.
	Component with Compatible License	Find a component that can be licensed in a manner that is compatible with the intended use.
	Create collective	Make sure the work is considered a collective that includes the component.
	Ask for exception	Request the licensor to give you an exception to one or more conditions imposed by the license. Results in the Exception Pattern, above.
	Ask for clarification	Request the licensor to clarify her interpretation of any ambiguous or contentious parts of the license. Results in the Clarification Pattern, above.

Figure 6.1 Identified patterns to address the license-mismatch problem [22].

The authors then discussed in detail four of those patterns including the intent, motivation, applicability, advantages, disadvantages, and known uses. Sadly the "model" the authors developed is merely predicate calculus (first-order logic) and not an automated system nor easily accessible to laypersons. (A little poorly, the authors were lax in the use of their language and confound copyright law and patent law in the introductory sections.) The future work section leaves much to be desired because it does not clearly outline directions forward.

Microsoft Research India and the National Institute of Technology Karnataka published "An Empirical Study of License Violations in Open Source Projects" in 2012 [70]. Using the now-defunct Google Code project hosting, 1423 projects were analyzed. "We observed a lack of proper use of the acceptor license in 3 out of the 4 cases of violations," noting that the downstream developer was the source of the fault. When comparing project activity (how actively maintained

a project is) with code reuse, the authors found that "projects that were actively developed and updated were reused more frequently and this is true for both corporate firms, as well as the open source world."

As recently as 2015, research has been conducted by Wu et al. [71] on the evolution of the licenses specified in the header of each file, with the explicit goal of finding license inconsistencies. They categorize the evolution of licenses as a license addition/removal, upgrade/downgrade, or change. These categorizations are then used to judge whether the new modification/evolution of the license results in an inconsistency. The authors used a series of tools (CCFinder [72] and Ninka [8]) to identify code clones and licenses. They found that 7.2 percent of file groups "contain one or more license inconsistencies." Notably the authors remark, "It is not a trivial task to find the repositories of these upstream projects."

In 2016, the SWAT–SOCCER Labs at Polytechnique Montréal published the paper "On the Detection of Licenses Violations in the Android Ecosystem." The group found that the most common licenses used for project releases were: GPL 3 (35%), Apache 2 (24%), and MIT (12%). "Out of the 857 studied apps, we found 17 apps with clear license violations." Most of these violations were the result of the GPL and another license conflicting. Interestingly, "licenses violations persist through multiple releases of the apps before they are eventually resolved," which took "19 releases [on] average" to fix [73].

"Automating the license compatibility process in open source software with SPDX" was published in 2017 by Kapitsaki el al. [13] The authors created SPDX Violation Tools to assist in the detection of license compatibility issues using a tool that looks at Software Package Data Exchange (SPDX) files of various projects. They devised a compatibility algorithm to surface license violations, which employed a directed acyclic (hierarchical) license compatibility graph. The algorithm traversed the graph to compare the licenses in use in a given project and dependencies. This approach is a rather precarious choice because license compatibility is not strictly hierarchical (although the authors did make some concessions for this fact) nor solely acyclic. For example, two licenses can be mutually compatible with one another. Sadly, the results found showed that "a

significant number of projects contain violations (11 out of 20 packages or 55%)" which included such popular projects as Hadoop, FileZilla, HandBrake.

Chapter 7

7.1 Conclusion

This research investigated the juncture between the fields of law and Software Engineering with a specific focus on incompatibilities between various Free/Libre and Open Source Licenses. License Violations are defined as the misuse of two or more licenses, where one (or more) license conditions disallow this combination. This study went beyond existing research to merely finding a new technique to detect license violations, and considered for the first time new transitive types of incompatibilities. License compatibility is widely misunderstood and poorly named as compatibility is unidirectional — that is not always reversible. To help alleviate this difficulty, three specific types of violations were outlined: Type 1 Simple Violations, Type 2 Embedded Transitive Violations, Type 3 Compound Transitive Violations. Concrete examples of Types 1 and 2 were found and analyzed.

A total of 131 996 Type 1 Simple Violations were found, and of those, the most prevalent pairing was Apache 2 ▸ GPL 2 with 65 105 examples discovered (49%). Embedded Transitive Violations were numerous with 281 744 found, and of those, the most common triple was EPL 1 ▸ Apache 2 ▸ GPL 3 with 254 447 examples (90%). Type 3 Compound Transitive Violations were critically examined and the found results were discussed due to the complexities of license terms in the found triples, however, a concrete example was outlined and expanding the license compatibility ontology is needed.

The primary research statement of this dissertation has been fulfilled; we were able to show that Oswaldo can detect license violations of both direct and transitive types using a flexible and extensible approach. However, the second research statement — set to determine which violation type is the most occurrent in the wild, and thus the most problematic — cannot truly be analyzed because of the lack of acceptable results for Type 3. With that said, Type 2 Embedded Transitive violations are indeed more numerous and widespread than classical Type 1 Simple Violations.

Nevertheless, both simple and transitive violations were shown to be a current problem in FLOSS. The detection of these violations is useful for software developers who use diversely-licensed software and may want to detect violations after-the-fact, legal counsel who create proactive guidelines for groups and organizations, as well as researchers interested in the interplay and incompatibility of open source licenses in use in the community on an internet scale. This research has contributed a novel technique to detect license violations and uniquely expanded upon the problem of license incompatibility by introducing transitive violations and exploring their attributes in thousands of open source projects.

Overall, given the reality that Free/Libre and Open Source Software is not going away anytime soon. Its legal complexities are something to contend with, yet manageable with proper tools and processes. When Richard Stallman, the creator of FLOSS, spoke at a conference in Montréal in 2017 [74], he argued that FLOSS should be used as a check-and-balance — a tool — to counter an overreaching state or wayward enterprise akin to how functional democracies view freedom of the press and their role in keeping politicians and powerful societal actors honest, held accountable, or at the very least, equipping the public to make informed decisions. This idea of open source as a check-and-balance has previously been dismissed as an example of Richard Stallman's eccentric frivolity. But recent events where professional journalists revealed that Uber programmed their mobile app to act in very dubious and unlawful ways [75], [76], have given more weight and relevance of his ideas. Viewable and modifiable source code, which is not at all incompatible with many tech companies' business plans, could be such a safeguard for the future.

7.2 Future Work

First, Oswaldo only modelled six out of twenty-one most-popular open source licenses, and thus may be missing pertinent incompatible license relationships. This expansion of the ontology and further investigation is left to future researchers.

Oswaldo, as a tool, could be further automated. For example, constantly scan the Maven data set for updates and incrementally update the triplestore with new facts. (This could be accomplished with SWRL rules, but as mentioned previously, Virtuoso Server does not support this feature.)

Additional work could then be done to use the twelve patterns from German & Hassan [22] to suggest ways to resolve these license violations.

The ontology in Oswaldo simplifies the relationship between two licenses down to a binary: compatible or incompatible. This was done to capture the most common usage scenario for a developer; using a dependency in a project and then distributing and running said project. An open source license has many grants, and not every developer uses the source code in the same way. In future, our ontology should be further developed to express compatibility/incompatibility for the five use cases outlined previously: 1. Linking, 2. Fork, 3. Subclass, 4. Intercommunication Protocol, 5. Plugin [22], as well as expanded for other common license grants, e.g. source code is publicly available, or license is distributed alongside the product [27].

This expansion would alleviate the problem with our current definition of "compatibleWith." Currently, the LGPL is "compatibleWith" another license if the use case is restricted to linking only. The definition used in this thesis may then miss some violations of the LGPL terms.

Recently, a license violation metric was developed as part of a trustworthy measure of software projects combining this research along with my labmates' research on bug and security vulnerability prevalence as well as software build systems [46]. The goal of such a metric is to help programmers choose a trustworthy dependency among many options.

No detailed survey has been conducted of the developer community regarding 'FLOSS literacy.' First a definition of FLOSS literacy would need to be outlined, such as how much a programmer knows: about the proper use of licenses; what are the defining principles of FLOSS, and what are the common basic requirements of FLOSS reuse. From casual conversations with fellow developers in my professional life, I have heard various myths and misunderstandings regarding FLOSS and licenses. It is a complex topic not easily distilled and disseminated throughout various communities (e.g. new graduates or seasoned developers using the GPL, or bloggers using Creative Commons). A survey should be devised to measure how 'literate' a person is, and further elaborated into how literate various groups are. Based on these results more concerted efforts can

be directed toward educating these individuals and communities, thus reducing license violations as well as generally promoting FLOSS.

In a certain light, this dissertation is a small admission that our self-made systems are somewhat irrational and do not necessarily match initial — nor evolving — intentions. Recent work by An et el. [3] has investigated the use of code snippets from Stack Overflow in open source Android projects. Such research is interesting as an intellectual pursuit but does nothing to change the status quo. Developers will continue to post snippets online intended to be unreservedly reused and other developers will continue to employ those fragments in their projects. These developers will do so being fully conscious that they are not adhering to the licensing conditions, but are most probably adhering to the original intent of posting source code online; to share and share alike. As an aside, I personally would love to see Stack Overflow change their policy for source code posted on their site to be free of any restrictions whatsoever (be that a dedication to the public domain, where applicable, or use of the so-called "Unlicense" [77]). Stack Overflow is likely hesitant to change their policy because of many copycat sites reusing their content for ad views, as well as the concern over their ranking of search results, if other sites reuse the same content. Similarly, this research does not affect the status quo currently. It is up to developers to fix their own mistakes. The old adage comes to mind: "you can lead a horse to water, but you can't make it drink." This fixing must be undertaken by the community in future.

7.3 Publications

Ellis Eghan, Sultan Alqahtani, Christopher Forbes, and Juergen Rilling, **"API Trustworthiness: An Ontological Approach for Software Library Adoption,"** 26th Software Quality Journal (SQJ) on Trustworthy Systems and Software, 2018 *(accepted)*.

Iman Keivanloo, Christopher Forbes, Aseel Hmood, Mostafa Erfani, Christopher Neal, George Peristerakis, and Juergen Rilling, **"A Linked Data Platform for Mining Software Repositories,"** 9th Working Conference on Mining Software Repositories (MSR), 2012.

Iman Keivanloo, Christopher Forbes, and Juergen Rilling, **"Similarity Search Plug-in: Clone Detection Meets Internet-scale Code Search,"** 4th ICSE Workshop on Search-Driven Development: Users, Infrastructure, Tools and Evaluation (SUITE), 2012.

Christopher Forbes, Iman Keivanloo, and Juergen Rilling, **"When Open Source Turns Cold on Innovation — The Challenges of Navigating Licensing Complexities in New Research Domains,"** 34th International Conference on Software Engineering (ICSE), Poster Track, 2012.

Christopher Forbes, Iman Keivanloo, and Juergen Rilling, **"Doppel-Code: A Clone Visualization Tool for Prioritizing Global and Local Clone Impacts,"** 36th IEEE International Computer Software and Applications Conference (COMPSAC), 2012.

Iman Keivanloo, Christopher Forbes, Juergen Rilling, and Philippe Charland, **"Towards Sharing Source Code Facts Using Linked Data,"** 3rd ICSE Workshop on Search-Driven Development: Users, Infrastructure, Tools and Evaluation (SUITE), 2011.

7.4 Bibliography

[1] R. Pressman, Software Engineering: A Practitioner's Approach, New York, NY: McGraw-Hill, 2010.

[2] M. J. Hawthorne and D. E. Perry, "Software Engineering Education in the Era of Outsourcing, Distributed Development, and Open Source Software: Challenges and Opportunities," in *27th International Conference on Software Engineering*, Saint Louis, MO, USA, 2005.

[3] L. An, O. Mlouki, F. Khomh and G. Antoniol, "Stack Overflow: A Code Laundering Platform?," in *International Conference on Software Analysis, Evolution, and Reengineering*, Klagenfurt, Austria, 2017.

[4] S. Weber, The Success of Open Source, Cambridge, MA, USA: Harvard University Press, 2004.

[5] Free Software Foundation, "Violations of the GNU Licenses," Free Software Foundation, 2 March 2017. [Online]. Available: https://www.gnu.org/licenses/gpl-violation.en.html. [Accessed 2 June 2017].

[6] Software Freedom Law Center, "BusyBox Developers Agree to End GPL Lawsuit Against Verizon," Software Freedom Law Center, 17 March 2008. [Online]. Available: https://www.softwarefreedom.org/news/2008/mar/17/busybox-verizon/. [Accessed 19 June 2017].

[7] R. Gobeille, "The FOSSology Project," in *International Working Conference on Mining Software Repositories* , Leipzig, Germany, 2008.

[8] D. M. German, Y. Manabe and K. Inoue, "A Sentence-matching Method for Automatic License Identification of Source Code Files," in *International Conference on Automated Software Engineering*, Antwerp, Belgium, 2010.

[9] G. M. Kapitsaki, N. D. Tselikas and I. E. Foukarakis, "An Insight into License Tools for Open Source Software Systems," *The Journal of Systems and Software,* vol. 102, pp. 72-87, 2015.

[10] H. Armijn, K. Trygve Kalleberg, R. Vermaas and E. Dolstra, "Finding Software License Violations Through Binary Code Clone Detection," *International Working Conference on Mining Software Repositories,* May 2011.

[11] B. Smith, "FSF Settles Suit Against Cisco," Free Software Foundation, 20 May 2009. [Online]. Available: https://www.fsf.org/news/2009-05-cisco-settlement.html. [Accessed 19 June 2017].

[12] J. Brodkin, "VMware alleged to have violated Linux's open source license for years," Ars Technica, 6 March 2015. [Online]. Available: https://arstechnica.com/tech-policy/2015/03/vmware-alleged-to-have-violated-linuxs-open-source-license-for-years/. [Accessed 19 June 2017].

[13] G. M. Kapitsaki, F. Kramer and N. D. Tselikas, "Automating the License Compatibility Process in Open Source Software with SPDX," *Journal of Systems and Software,* vol. 131, pp. 386-401, 2017.

[14] I. Keivanloo, C. Forbes and J. Rilling, "Towards Sharing Source Code Facts Using Linked Data," in *International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, Waikiki, USA, 2011.

[15] C. Forbes, I. Keivanloo and J. Rilling, "Doppel-Code: A Clone Visualization Tool for Prioritizing Global and Local Clone Impacts," in *International Conference on Computer Software and Applications*, Izmir, Turkey, 2012.

[16] C. Forbes, I. Keivanloo and J. Rilling, "When Open Source Turns Cold on Innovation - The Challenges of Navigating Licensing Complexities in New Research Domains," in *International Conference on Software Engineering (ICSE)*, Zürich, Switzerland, 2012.

[17] A. M. St. Laurent, Understanding Open Source and Free Software Licensing, Sebastopol, California: O'Reilly Media, 2004.

[18] P. Miller, R. Styles and T. Heath, "Open Data Commons, A License for Open Data," in *Workshop on Linked Data on the Web (LDOW)*, Beijing, China, 2008.

[19] I. Keivanloo, C. Forbes and J. Rilling, "Similarity Search Plug-in: Clone Detection Meets Internet-scale Code Search," in *ICSE Workshop on Search-Driven Development - Users, Infrastructure, Tools and Evaluation*, Zurich, Switzerland, 2012.

[20] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis and J. Rilling, "A Linked Data Platform for Mining Software Repositories," in *International Conference on Mining Software Repositories (MSR)*, Zurich, Switzerland, 2012.

[21] Software Freedom Law Center, "Best Buy, Samsung, Westinghouse, And Eleven Other Brands Named In SFLC Lawsuit," Software Freedom Law Center, 14 December 2009. [Online]. Available: https://www.softwarefreedom.org/news/2009/dec/14/busybox-gpl-lawsuit/. [Accessed 2 July 2017].

[22] D. M. German and A. E. Hassan, "License Integration Patterns: Addressing License Mismatches in Component-based Development," in *Proceedings of the 31st International Conference on Software Engineering*, Vancouver, Canada, 2009.

[23] S. Williams and R. M. Stallman, Free as in Freedom, Boston, MA: Free Software Foundation, 2010.

[24] R. Stallman, Free Software, Free Society: Selected Essays of Richard M. Stallman, Boston, MA: Free Software Foundation, 2015.

[25] D. R. Booth, Peer Participation and Software: What Mozilla Has to Teach Government, Cambridge, MA: The MIT Press, 2010.

[26] Open Source Initiative, "History of the OSI," September 2012. [Online]. Available: https://opensource.org/history. [Accessed 10 December 2017].

[27] Open Source Initiative, "The Open Source Definition," 22 March 2007. [Online]. Available: https://opensource.org/osd. [Accessed 10 December 2017].

[28] L. Rosen, Open Source Licensing: Software Freedom and Intellectual Property Law, Upper Saddle River, New Jersey: Prentice Hall.

[29] GitHub, Inc., "Choose a License," [Online]. Available: https://choosealicense.com/appendix/. [Accessed 9 10 2017].

[30] Ruby Together, "RubyGems.org | your community gem host," [Online]. Available: https://rubygems.org/. [Accessed 3 January 2018].

[31] The Apache Software Foundation, "Apache Maven Project," 31 December 2017. [Online]. Available: https://maven.apache.org/. [Accessed 3 January 2018].

[32] The CocoaPods Dev Team, "CocoaPods," [Online]. Available: https://cocoapods.org/. [Accessed 3 January 2018].

[33] O. Seneviratne, L. Kagal, D. Weitzner, H. Abelson, T. Berners-Lee and N. Shadbolt, "Detecting Creative Commons License Violations on Images on the World Wide Web," in *International World Wide Web Conference (WWW)*, Madrid, Spain, 2009.

[34] Oxford University Press, Oxford Dictionary of English, Oxford: Oxford University Press, 2010.

[35] C. K. Roy, J. R. Cordy and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming,* vol. 74, no. 7, pp. 470-495, 2009.

[36] R. W. Gomulkiewicz, "Open Source License Proliferation: Helpful Diversity or Hopeless Confusion," *Washington University Journal of Law & Policy,* vol. 30, no. 1, pp. 261-291, 2009.

[37] Gouvernement du Québec, "Licence - Forge gouvernementale," 24 January 2018. [Online]. Available: https://forge.gouv.qc.ca/licence/. [Accessed 2 February 2018].

[38] Free Software Foundation, "Various Licenses and Comments About Them," GNU Project, 4 April 2017. [Online]. Available: https://www.gnu.org/licenses/license-list.html. [Accessed 22 July 2017].

[39] T. R. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition,* vol. 5, no. 2, pp. 199-220, June 1993.

[40] World Wide Web Consortium, "W3C Semantic Web Activity," 11 December 2013. [Online]. Available: https://www.w3.org/2001/sw/. [Accessed 24 February 2018].

[41] W3C, "Resource Description Framework (RDF): Concepts and Abstract Syntax," 10 February 2004. [Online]. Available: https://www.w3.org/TR/rdf-concepts/. [Accessed 4 January 2018].

[42] W3C, "SPARQL Query Language for RDF," 15 January 2008. [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/. [Accessed 4 January 2018].

[43] T. Berners-Lee, "Linked Data," 27 July 2006. [Online]. Available: https://www.w3.org/DesignIssues/LinkedData.html. [Accessed 4 January 2018].

[44] W3C, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," 21 May 2004. [Online]. Available: https://www.w3.org/Submission/SWRL/. [Accessed 4 January 2018].

[45] G. Bavota, A. Ciemniewska, I. Chulani, A. De Nigro, M. Di Penta, D. Galletti, R. Galoppini, T. F. Gordon, P. Kedziora, I. Lener, F. Torelli, R. Pratola, J. Pukacki, Y. Rebahi and S. García Villalonga, "The MARKet for Open Source: An Intelligent Virtual Open Source Marketplace," in *Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Antwerp, Belgium, 2014.

[46] E. Eghan, S. Alqahtani, C. Forbes and J. Rilling, "API Trustworthiness: An Ontological Approach for Software Library Adoption," *Software Quality Journal on Trustworthy Systems and Software,* 2018 (accepted).

[47] A. Hmood, I. Keivanloo and J. Rilling, "SE-EQUAM — An Evolvable Quality Metamodel," in *Computer Software and Applications Conference Workshops (COMPSACW)*, Izmir, Turkey, 2012.

[48] S. Alqahtani, E. Eghan and J. Rilling, "SV-AF — A Security Vulnerability Analysis Framework," in *International Symposium on Software Reliability Engineering (ISSRE)*, Ottawa, Canada , 2016.

[49] Software Freedom Law Center, "Motion Against Westinghouse Digital Electronics in GPL Compliance Lawsuit," Software Freedom Law Center, 7 June 2010. [Online]. Available: https://www.softwarefreedom.org/news/2010/jun/07/motion-against-westinghouse-digital-electronics-gp/. [Accessed 2 July 2017].

[50] R. Wille, "Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts," *Ordered Sets,* vol. 83, pp. 445-470, 1982.

[51] Y. Hasebe and K. Kuroda, "Extraction of English Ditransitive Constructions Using Formal Concept Analysis," *Pacific Asia Conference on Language, Information and Computation,,* vol. 23, no. 2, p. 678–685, December 2009.

[52] Y. Hasebe and K. Kuroda, "RubyFCA: Command line tool for Formal Concept Analysis written in Ruby," GitHub Inc., 4 May 2017. [Online]. Available: https://github.com/yohasebe/rubyfca. [Accessed 29 December 2017].

[53] B. Ganter, "Two Basic Algorithms in Concept Analysis," 1984.

[54] Apache Software Foundation, "Apache License v2.0 and GPL Compatibility," Apache Software Foundation, 2017. [Online]. Available: https://www.apache.org/licenses/GPL-compatibility.html. [Accessed 2 July 2017].

[55] European Commission, "EUPL compatible open source licences," European Commission, 24 May 2017. [Online]. Available: https://joinup.ec.europa.eu/community/eupl/og_page/eupl-compatible-open-source-licences. [Accessed 22 July 2017].

[56] B. M. Kuhn, A. K. Sebro and D. Gingerich, "Chapter 10 The Lesser GPL," Free Software Foundation & Software Freedom Law Center, 25 October 2016. [Online]. Available: https://copyleft.org/guide/comprehensive-gpl-guidech11.html. [Accessed 10 November 2017].

[57] Free Software Foundation, "GNU's Bulletin, vol. 1," GNU Project, 10 January 1991. [Online]. Available: https://www.gnu.org/bulletins/bull10.html#SEC6. [Accessed 4 November 2017].

[58] M. A. Musen, "The Protégé Project: A Look Back and a Look Forward," *AI Matters,* vol. 1, no. 4, pp. 4-12, 2015.

[59] C. Forbes, "markos-oss-licenses.owl," 22 April 2017. [Online]. Available: https://github.com/chrisjf/markos/blob/master/MARKOS/markos-oss-licenses.owl. [Accessed 22 April 2017].

[60] OpenLink Software, "OpenLink Virtuoso," [Online]. Available: https://virtuoso.openlinksw.com/. [Accessed 3 January 2018].

[61] C. Jensen and W. Scacchi, "License Update and Migration Processes in Open Source Software Projects," in *International Conference on Open Source Systems*, Salvador, Brazil, 2011.

[62] Mozilla Foundation, "Mozilla Public License Version 1.1," 1999. [Online]. Available: https://www.mozilla.org/en-US/MPL/1.1/. [Accessed 9 December 2017].

[63] Mozilla Foundation, "Mozilla Public License Version 2.0," 3 January 2012. [Online]. Available: https://www.mozilla.org/en-US/MPL/2.0/. [Accessed 9 December 2017].

[64] S. Shankland, "Torvalds: No GPL 3 for Linux," 30 January 2006. [Online]. Available: https://www.cnet.com/news/torvalds-no-gpl-3-for-linux/. [Accessed 9 December 2017].

[65] Free Software Foundation, "GNU Lesser General Public License Version 3," 29 June 2007. [Online]. Available: https://www.gnu.org/licenses/lgpl-3.0.html. [Accessed 9 December 2017].

[66] Mozilla Foundation, "License Boilerplate," 27 August 2015. [Online]. Available: https://website-archive.mozilla.org/www.mozilla.org/mpl/MPL/boilerplate-1.1/. [Accessed 19 January 2018].

[67] A. Kauffmann, "La nouvelle version 2 de la Mozilla Public License tend vers l'unité," Framasoft, 22 January 2012. [Online]. Available: https://framablog.org/2012/01/22/pozilla-public-license-version-2/. [Accessed 19 January 2018].

[68] A. Monden, S. Okahara, Y. Manabe and K. Matsumoto, "Guilty or Not Guilty: Using Clone Metrics to Determine Open Source Licensing Violations," *IEEE Software,* vol. 28, no. 2, pp. 42 - 47, March 2011.

[69] M. Di Penta, D. M. German, Y.-G. Guéhéneuc and G. Antoniol, "An Exploratory Study of the Evolution of Software Licensing," in *International Conference on Software Engineering*, Cape Town, South Africa, 2010.

[70] A. Mathur, H. Choudhary, P. Vashist, W. Thies and S. Thilagam, "An Empirical Study of License Violations in Open Source Projects," in *Software Engineering Workshop*, Heraclion, Greece, 2012.

[71] Y. Wu, Y. Manabe, T. Kanda, D. M. German and K. Inoue, "A Method to Detect License Inconsistencies in Large-Scale Open Source Projects," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, Florence, Italy, 2015.

[72] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering,* vol. 28, no. 7, pp. 654-670, 2002.

[73] O. Mlouki, F. Khomh and G. Antoniol, "On the Detection of Licenses Violations in the Android Ecosystem," in *International Conference on Software Analysis, Evolution, and Reengineering*, Suita, Japan, 2016.

[74] McGill University, "Richard Stallman "Freeing Science — From Software to Medicine" Lecture," 12 May 2017. [Online]. Available: https://www.mcgill.ca/medicine/channels/event/richard-stallman-freeing-science-software-medicine-lecture-267808. [Accessed 3 January 2018].

[75] M. Isaac, "How Uber Deceives the Authorities Worldwide," New York Times, 3 March 2017. [Online]. Available: https://www.nytimes.com/2017/03/03/technology/uber-greyball-program-evade-authorities.html. [Accessed 3 January 2018].

[76] A. Hern, "Uber employees 'spied on ex-partners, politicians and Beyoncé'," The Guardian, 13 December 2016. [Online]. Available: https://www.theguardian.com/technology/2016/dec/13/uber-employees-spying-ex-partners-politicians-beyonce. [Accessed 3 January 2018].

[77] "Unlicense Yourself: Set Your Code Free," [Online]. Available: https://unlicense.org/. [Accessed 2 January 2018].

[78] The Apache Software Foundation, "Apache License v2.0 and GPL Compatibility," [Online]. Available: https://www.apache.org/licenses/GPL-compatibility.html. [Accessed 22 July 2017].

7.5 Appendices

7.5.1 TripleConstructor.java

```
package ca.concordia.cs.aseg.secont.virtuoso;

import java.io.FileNotFoundException;

import virtuoso.jena.driver.VirtGraph;
import virtuoso.jena.driver.VirtuosoQueryExecution;
import virtuoso.jena.driver.VirtuosoQueryExecutionFactory;

import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.graph.Triple;
import com.hp.hpl.jena.query.QuerySolution;
import com.hp.hpl.jena.query.ResultSet;
import com.hp.hpl.jena.rdf.model.RDFNode;

public class TripleConstructor {
    private static String materializationGraphIRI;

    private static String graphCodeIRI = "http://code-data.com";
    private static String graphHistoryIRI = "http://history-data.com";
    private static String graphBuildIRI = "http://build-data.com";
    private static String graphSecurityIRI = "http://sevont-data.com";
    private static String graphMainIRI = "http://svaf-data.com";

    private static String graphSchemaIRI = "http://svaf-schema.com";

    private static String graphRulesIRI = "svaf-rule-sets";

    public static String url = "jdbc:virtuoso://slicer:1111";
    public static String password = "dba";
    public static String username = "dba";

    public static void main(String[] args) throws FileNotFoundException {
        //inferLicenses();
        inferTransitiveDependencies();
    }

    public static VirtGraph getConnectionToStore() {
        VirtGraph set = new VirtGraph(materializationGraphIRI, url, username,
                password);
        return set;
    }

    private ResultSet getInferredTriples(VirtGraph set, String query) {
        VirtuosoQueryExecution vqe = VirtuosoQueryExecutionFactory.create(
```

```
                query, set);
        ResultSet results = vqe.execSelect();
        return results;
    }

    private void materializeInferredTriples(VirtGraph set, ResultSet results,
            boolean isLiteralObject) {
        while (results.hasNext()) {
            QuerySolution result = results.nextSolution();
            RDFNode s = result.get("S");
            RDFNode p = result.get("P");
            RDFNode o = result.get("O");
            Node S = Node.createURI(s.toString());
            Node P = Node.createURI(p.toString());
            Node O;
            if (isLiteralObject)
                O = Node.createLiteral(o.toString());
            else
                O = Node.createURI(o.toString());
            set.add(new Triple(S, P, O));
        }
    }

    private static void inferLicenses() {
        materializationGraphIRI = "http://svaf-data.com";

        String query = // "DEFINE input:inference '"+graphRulesIRI+"'\n"
        "PREFIX markosLic:<http://www.markosproject.eu/ontologies/licenses#>\n"
                + "PREFIX markos:<http://www.markosproject.eu/ontologies/oss-
                 licenses#>\n"
                + "PREFIX maven: <http://aseg.cs.concordia.ca/segps/ontologies/
                 system-specific/2015/02/maven.owl#>\n"
                + "CONSTRUCT {?s markosLic:coveringLicense ?l.}\n" + "FROM <"
                + graphBuildIRI + ">\n" + "FROM <" + graphSchemaIRI + ">\n"
                + "where {\n" + "?l a ?class.\n"
                + "?class rdfs:subClassOf+ markos:OpenSourceLicenseTemplate.\n"
                + "?l rdfs:seeAlso ?url1.\n"
                + "?s maven:hasLicenseUrl ?url2.\n"
                + "FILTER(?url2=str(?url1)).\n" + "}";

        // System.out.println(query);

        VirtGraph set = TripleConstructor.getConnectionToStore();
        System.out.println("graph.getCount() before = " + set.getCount());
        System.out
                .println("materializing inferred Maven licenses to the graph");
        TripleConstructor constructor = new TripleConstructor();
        ResultSet resultSet = constructor.getInferredTriples(set, query);
        constructor.materializeInferredTriples(set, resultSet, false);
        System.out.println("graph.getCount() after = " + set.getCount());
        System.out.println("Done !!");
```

```
    }

    private static void inferTransitiveDependencies() {
        materializationGraphIRI = "http://build-data.com";

        String transOptionalQuery =
                "PREFIX build: <http://aseg.cs.concordia.ca/segps/ontologies/
                 domain-specific/2015/02/build.owl#>\n"
               + "PREFIX maven: <http://aseg.cs.concordia.ca/segps/ontologies/
                system-specific/2015/02/maven.owl#>\n"
               + "SELECT ?projC ?projB\n"
               + "FROM <"+ graphBuildIRI + ">\n"
               + "where {\n"
               +"?link build:hasDependencySource ?projC.\n"
               +"?link build:hasDependencyTarget ?projA.\n"
               +"?link build:hasNumberOfExclusions \"0\".\n"

               +"?projC build:hasOptionalBuildDependencyOn ?projA.\n"
               +"?projA build:hasNonOptionalBuildDependencyOn ?projB.\n"
               + "}";

        String transNonOptionalQuery =
                "PREFIX build: <http://aseg.cs.concordia.ca/segps/ontologies/
                 domain-specific/2015/02/build.owl#>\n"
               + "PREFIX maven: <http://aseg.cs.concordia.ca/segps/ontologies/
                system-specific/2015/02/maven.owl#>\n"
               + "SELECT ?projC ?projB\n"
               + "FROM <"+ graphBuildIRI + ">\n"
               + "where {\n"
               +"?link build:hasDependencySource ?projC.\n"
               +"?link build:hasDependencyTarget ?projA.\n"
               +"?link build:hasNumberOfExclusions \"0\".\n"

               +"?projC build:hasNonOptionalBuildDependencyOn ?projA.\n"
               +"?projA build:hasNonOptionalBuildDependencyOn ?projB.\n"
               + "}";
        System.out.println(transOptionalQuery);
        System.out.println(transNonOptionalQuery);

        /*VirtGraph set = TripleConstructor.getConnectionToStore();
        System.out.println("graph.getCount() before = " + set.getCount());
        System.out
                .println("materializing inferred Maven transitive dependencies
                 to the graph");
        TripleConstructor constructor = new TripleConstructor();
        ResultSet resultSet = constructor.getInferredTriples(set, query);
        constructor.materializeInferredTriples(set, resultSet, false);
        System.out.println("graph.getCount() after = " + set.getCount());*/
        System.out.println("Done !!");
    }
}
```

7.5.2 Type1Analysis.swift

```
//
//  Type1Analysis.swift
//  Violations Analysis
//
//  Created by Christopher Forbes on 2017-07-22.
//  Copyright © 2017 Christopher Forbes. All rights reserved.
//

import Foundation

private struct SimpleViolation {
    let link: String
    let repository1: String
    let repository2: String
    let license1: String
    let license2: String
}

// This code lists all the violation pairs i.e. so you can see which combo of
// licenses is the most problematic.

class Type1Analysis {

    // link,    repository1,    repository2,    license1,   license2
    var counts: Dictionary<String, Int> = [:]

    func start() {
        DispatchQueue.global(qos: .userInitiated).async {
            self.gatherViolations()
            self.printResults()
        }
    }

    func gatherViolations() {
        let path = "/Users/chris/Documents/Projects/Violations Analysis/Results
         Unzipped/Type 1.csv"

        let url = URL(fileURLWithPath: path)
        guard let data = try? Data(contentsOf: url) else { return }
        guard let content = String(data: data, encoding: .ascii) else
         { return }

        var rows = content.split(separator: "\n").map(String.init)
        rows.remove(at: 0) // remove the first "header" row

        for row in rows {
            let columns = row.components(separatedBy: ",")
```

```
            let license1raw = columns[3]
            let license2raw = columns[4]

            let license1 = removeQuotationMarks(from: license1raw)
            let license2 = removeQuotationMarks(from: license2raw)

            let violation = SimpleViolation(link: columns[0], repository1:
             columns[1], repository2: columns[2], license1: String(license1),
             license2: String(license2))

            // Count pairs
            let pair = "\"\(violation.license1) into \(violation.license2)\""
            var count: Int
            if let previous = counts[pair] {
                count = previous + 1
            } else {
                count = 1
            }
            counts.updateValue(count, forKey: pair)
        }
    }

    func removeQuotationMarks(from licenseURL: String) -> String {
        // "http://www.markosproject.eu/ontologies/oss-licenses#GPL-2.0"
        let startIndex = licenseURL.index(licenseURL.startIndex, offsetBy: 53)
        let endIndex = licenseURL.index(licenseURL.endIndex, offsetBy: -1)
        let license = licenseURL[startIndex..<endIndex]
        return String(license)
    }

    func printResults() {
        for pair in counts {
            print("\(pair.key),\(pair.value)")
        }
    }

}
```

7.5.3 Type2Analysis.swift

```
//
//  Type2Analysis.swift
//  Violations Analysis
//
//  Created by Christopher Forbes on 2017-08-24.
//  Copyright © 2017 Christopher Forbes. All rights reserved.
//

import Foundation

private struct EmbeddedViolation {
    let linkA: String
    let linkB: String
    let repository1: String
    let repository2: String
    let repository3: String
    let license1: String
    let license2: String
    let license3: String
}

// This code lists all the violation triples
// i.e. so you can see which combo of licenses is the most problematic.

class Type2Analysis {

    //
     "linkA","repository1","repository2","linkB","repository3","license1","lice
     nse2","license3"
    var counts: Dictionary<String, Int> = [:]

    func start() {
        DispatchQueue.global(qos: .userInitiated).async {
            self.gatherViolations()
            self.printResults()
        }
    }

    func gatherViolations() {
        let path = "/Users/chris/Documents/Projects/Violations Analysis/Results
         Unzipped/Type 2.csv"

        let url = URL(fileURLWithPath: path)
        guard let data = try? Data(contentsOf: url) else { return }
        guard let content = String(data: data, encoding: .ascii) else
         { return }
```

```
        var rows = content.split(separator: "\n").map(String.init)
        rows.remove(at: 0) // remove the first "header" row

        for row in rows {
            let columns = row.components(separatedBy: ",")

            let license1raw = columns[5]
            let license2raw = columns[6]
            let license3raw = columns[7]

            let license1 = removeQuotationMarks(from: license1raw)
            let license2 = removeQuotationMarks(from: license2raw)
            let license3 = removeQuotationMarks(from: license3raw)

            let violation = EmbeddedViolation(linkA: columns[0],
                                              linkB: columns[3],
                                              repository1: columns[1],
                                              repository2: columns[2],
                                              repository3: columns[4],
                                              license1: license1,
                                              license2: license2,
                                              license3: license3)

            // Count triples
            let combo = "\"\(violation.license1) into \(violation.license2)
             into \(violation.license3)\""
            var count: Int
            if let previous = counts[combo] {
                count = previous + 1
            } else {
                count = 1
            }
            counts.updateValue(count, forKey: combo)
        }
    }

    func removeQuotationMarks(from licenseURL: String) -> String {
        // "http://www.markosproject.eu/ontologies/oss-licenses#GPL-2.0"
        let startIndex = licenseURL.index(licenseURL.startIndex, offsetBy: 53)
        let endIndex = licenseURL.index(licenseURL.endIndex, offsetBy: -1)
        let license = licenseURL[startIndex..<endIndex]
        return String(license)
    }

    func printResults() {
        for pair in counts {
            print("\(pair.key),\(pair.value)")
        }
    }

}
```

7.5.4 Type3Analysis.swift

```
//
//  Type3Analysis.swift
//  Violations Analysis
//
//  Created by Christopher Forbes on 2017-08-25.
//  Copyright © 2017 Christopher Forbes. All rights reserved.
//

import Foundation

private struct CompoundViolation {
    let linkA: String
    let linkB: String
    let repository1: String
    let repository2: String
    let repository3: String
    let license1: String
    let license2: String
    let license3: String
}

// This code lists all the violation triples
// i.e. so you can see which combo of licenses is the most problematic.

class Type3Analysis {

    //"linkA","repository1","repository3","linkB","repository2","license1","lic
     ense2","license3"
    var counts: Dictionary<String, Int> = [:]

    func start() {
        DispatchQueue.global(qos: .userInitiated).async {
            self.gatherViolations()
            self.printResults()
        }
    }

    func gatherViolations() {
        let path = "/Users/chris/Documents/Projects/Violations Analysis/Results
         Unzipped/Type 3.csv"

        let url = URL(fileURLWithPath: path)
        guard let data = try? Data(contentsOf: url) else { return }
        guard let content = String(data: data, encoding: .ascii) else
         { return }

        var rows = content.split(separator: "\n").map(String.init)
```

```
        rows.remove(at: 0) // remove the first "header" row

        for row in rows {
            let columns = row.components(separatedBy: ",")

            let license1raw = columns[5]
            let license2raw = columns[6]
            let license3raw = columns[7]

            let license1 = removeQuotationMarks(from: license1raw)
            let license2 = removeQuotationMarks(from: license2raw)
            let license3 = removeQuotationMarks(from: license3raw)

            let violation = CompoundViolation(linkA: columns[0],
                                              linkB: columns[3],
                                              repository1: columns[1],
                                              repository2: columns[4],
                                              repository3: columns[2],
                                              license1: license1,
                                              license2: license2,
                                              license3: license3)

            // Count triples
            let combo = "\"\(violation.license1) and \(violation.license2) into
             \(violation.license3)\""
            var count: Int
            if let previous = counts[combo] {
                count = previous + 1
            } else {
                count = 1
            }
            counts.updateValue(count, forKey: combo)
        }
    }

    func removeQuotationMarks(from licenseURL: String) -> String {
        // "http://www.markosproject.eu/ontologies/oss-licenses#GPL-2.0"
        let startIndex = licenseURL.index(licenseURL.startIndex, offsetBy: 53)
        let endIndex = licenseURL.index(licenseURL.endIndex, offsetBy: -1)
        let license = licenseURL[startIndex..<endIndex]
        return String(license)
    }

    func printResults() {
        for pair in counts {
            print("\(pair.key),\(pair.value)")
        }
    }

}
```