Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

ALEXANDER MARX
MASTER THESIS

EIN WERKZEUG ZUR BESCHREIBUNG UND BEWERTUNG VON CODE-KOMPONENTEN-BASIERTEN SOFTWAREARCHITEKTUREN

Eingereicht am 24. März 2015

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 24. März 2015

License

Erlangen, 24. März 2015

Abstract

The analysis and evaluation of component based software architectures is an important part of software engineering. In such a context it is not only crucial to analyze the performance, security etc. but also the compatibility between different licensed components of the architecture. For modelling and visualization of the software architecture, an architecture description language and an editor were initially developed in this thesis. The architecture description language allows the formal modeling of an architecture. The editor allows to display and edit the architecture. To include the possibility of automatic retrieval of a license from it's source code, this work connects a FOSSology Server via a REST-Interface.
A key aspect of the thesis at hand constitutes the analysis of license compatibility between the components. For this purpose existing approaches were evaluated and based on this, two license models for machine readable representation were developed. Those models were complemented by algorithms, which test the license compatibility. Finally, the functionality of both models and algorithms was examined by means of two example architectures and a comparison between the results was drawn.

Zusammenfassung

Die Analyse und Bewertung von komponentenbasierten Softwarearchitekturen ist ein wichtiger Teil der Softwareentwicklung. Entscheidend dabei ist nicht nur die Analyse auf Performanz, Sicherheit etc., sondern auch auf Lizenzkompatibilität zwischen den unterschiedlich lizenzierten Komponenten der Architektur. In dieser Arbeit wurde zunächst eine Architecture Description Language sowie ein Editor für die Modellierung und Visualisierung einer Softwarearchitektur entwickelt. Die Architecture Description Language erlaubt es, eine Architektur formal zu modellieren. Der Editor ermöglicht eine Darstellung und Veränderung der Architektur. Um eine Möglichkeit zu schaffen, die Lizenzen für eine Komponente der Architektur automatisch aus ihrem Source Code zu beziehen, wurde ein FOSSology Server über eine REST-Schnittstelle angebunden. Einen Schwerpunkt der vorliegenden Arbeit bildet die Analyse der Lizenzkompatibilität zwischen den Komponenten. Hierfür wurden bereits existierende Ansätze evaluiert und auf Basis deren zwei Modelle für die maschinenlesbare Darstellung von Softwarelizenzen entwickelt. Die Modelle wurden ergänzt durch Algorithmen, welche die Kompatibilität prüfen. Abschließend wurde die Funktionsweise der entwickelten Modelle und Algorithmen mit Hilfe von Beispielarchitekturen untersucht und ein Vergleich zwischen den Ergebnissen gezogen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ADL	Architecture Description Language
AGPL	GNU Affero General Public License
bSAM	Symbolic Alignment Matrix
ccREL	Creative Commons Rights Expression Language
CSV	Comma-Separated-Value
CTL	Corel Transactional License
DPMA	Das Deutsche Patent- und Markenamt
EPL	Eclipse Public License
FOSS	Free and Open Source Software
GNU GPL	GNU General Public License
GWT	Google Web Toolkit
IPC	Inter Process Communication
JSON	JavaScript Object Notation
LGPL	GNU Lesser General Public License
MS-PL	Microsoft Public License
MVP	Model-View-Presenter
OSD	Open Source Definition
OSI	Open Source Initiative
RDF	Ressource Description Framework
REL	Rights Expression Language
RNLS	Restricted Natural Language Statement
UI	User Interface

1 Einführung

Seit dem Beginn der Softwareentwicklung in den 1950er Jahren hat sich das Gebiet rasant weiterentwickelt und wurde dadurch auch immer komplexer. Heutzutage sind eine große Anzahl von Entwicklern, meist aufgeteilt in mehrere Teams, notwendig um ein professionelles Softwareentwicklungsprojekt durchzuführen. Ein Ansatz um die große Komplexität besser in den Griff zu bekommen und Softwareprojekte besser zu strukturieren, ist die sogenannte komponentenbasierte Softwareentwicklung. Bei dieser Art der Softwareentwicklung wird versucht ein System aus lose gekoppelten Komponenten zusammenzusetzen. Ein Vorteil dieser Methode ist, dass Komponenten wiederverwendet werden können, da diese unabhängig voneinander sind und nur über definierte Schnittstelle miteinander kommunizieren. (Vogel et al., 2009, S. 161-163) Um eine solche Wiederverwendung von Komponenten besser zu ermöglichen, können diese unter einer Open Source Lizenz für andere Entwickler freigegeben werden. Heutzutage werden immer mehr große Softwaresysteme auf diese Art entwickelt. Es wurde auch bereits ein großer Aufwand dahingehend investiert, Anwendungen zur Unterstützung der Modellierung und Verwaltung von Softwarearchitekturen zu entwickeln. Diese Arbeiten konzentrieren sich jedoch hauptsächlich auf die fachgerechte Umsetzung von Anforderungen, die Definition der Architektur in einer formalen Schreibweise, die Evaluierung einer bestehenden Architektur, die Dokumentation der Architektur und auf die weitergehende Analyse von Softwarearchitekturen bezüglich ihrer Performanz, Sicherheit, etc. Es gibt jedoch bisher nur wenig Ansätze, welche auch den rechtlichen Rahmen einer Softwarearchitektur betrachten. Durch die oben erwähnte Komponentenbasierte Softwareentwicklung entstehen gerade in diesem Bereich neue Herausforderungen. Eine davon ist die unterschiedliche Lizenzierung der verwendeten Komponenten und deren Auswirkung auf das Gesamtsystem. Durch die steigende Popularität von Open Source Software, insbesondere in Unternehmen, werden immer häufiger Softwaresysteme aus bereits vorher existierenden Open Source Komponenten erzeugt oder durch diese erweitert. Dies ist ein sehr komplexes Problem, welchem sich diese Arbeit widmen soll. Im folgenden Abschnitt werden die genauen Ziele der Arbeit erläutert.

1.1 Ziel der Masterarbeit

Diese Arbeit beschäftigt sich mit der Analyse der im vorherigen Abschnitt beschriebenen Softwaresysteme. Ziel ist es, einen Editor zu realisieren, welcher es ermöglicht, Softwaresysteme, bestehend aus einzelnen Komponenten, visuell als Graph darstellen zu können. Des Weiteren soll es möglich sein, für die einzelnen Komponenten und ihre Verbindungen relevante Informationen eintragen und bearbeiten zu können. Solche Informationen beinhalten z.B. den Namen der Komponente, eine aussagekräftige Beschreibung, sowie die Art der Verbindung zwischen den Komponenten. Ein weiterer wichtiger Punkt dabei ist die Beschreibung der verwendeten Lizenzen und im Besonderen Komponenten, welche unter einer Open Source Lizenz veröffentlicht wurden. Um die Lizenzen der einzelnen Komponenten herausfinden zu können, soll zunächst eine Analyse der Komponenten durchgeführt werden. Für diese Analyse stützt sich die Arbeit auf das *FOSSology* Projekt (Gobeille, 2008), welches für ein gegebenes Archiv alle Dateien auf Lizenzinformationen untersucht und alle gefundenen Lizenzen meldet. Anschließend soll es möglich sein, die modellierte Architektur auf Lizenzkonflikte zu untersuchen. Hierfür muss ein Algorithmus entwickelt werden, welcher verschiedene Open Source Lizenzen miteinander vergleichen und eine Aussage über deren Kompatibilität machen kann. Abschließend wird im Editor das Ergebnis der Analyse präsentiert.

2 Forschung

In diesem Kapitel sollen zunächst grundlegende Sachverhalte erläutert werden, welche für das weitere Verständnis dieser Arbeit unerlässlich sind. Hierzu gehört der Begriff der Free and Open Source Software (FOSS), die Beschreibung was eine Softwarelizenz ist und eine Einführung in FOSSology. Des Weiteren werden verwandte Arbeiten vorgestellt, die daraus abgeleiteten Anforderungen erläutert und die erarbeiteten Konzepte und Modelle behandelt.

2.1 Grundlagen

2.1.1 FOSS

Der Begriff FOSS bedeutet "Free and Open Source Software" und beschreibt eine Kombination von zwei Philosophien, nämlich der *Free Software* und der *Open Source Software*. Die Free Software ist die Ältere von beiden. Der Begriff wurde von Richard Stalmann ins Leben gerufen, als er das GNU-Projekt gegründet hatte. (*The GNU Project*, 2014) Die Idee hinter Free Software ist, dass der Begriff "free" (frei) in diesem Fall mit Freiheit und nicht mit "kostenlos" assoziiert werden soll. Richard Stalmann definiert ein Programm für einen Benutzer als Free Software, wenn:

- "Sie die Freiheit haben, das Programm auszuführen wie Sie möchten, für jeden Zweck;
- Sie die Freiheit haben, das Programm an Ihre Bedürfnisse anzupassen (um diese Freiheit in der Praxis umzusetzen, muss man Zugang zum Quellcode haben, denn Programmänderungen ohne Quellcode sind außerordentlich schwierig);
- Sie die Freiheit haben, Kopien weiterzuverbreiten, entweder gratis oder gegen eine Gebühr;

- Sie die Freiheit haben, modifizierte Programmversionen zu verbreiten, damit die Gemeinschaft von Ihren Verbesserungen profitieren kann."
  (*The GNU Project*, 2014)

Um die Förderung und Entwicklung von freier Software weiter voran zu bringen, wurde 1985 die *Free Software Foundation* gegründet. Diese ist unter anderem Hauptsponsor des GNU-Projektes und Herausgeber der *GNU General Public License* (GNU GPL) der weltweit beliebtesten Free Software Lizenz. (*Free Software Foundation*, 2015)
Da der Begriff *Free Software* für viele verwirrend war und weil die Free Software Foundation eine eher unternehmensfeindliche Meinung vertritt, wurden erste Überlegungen durchgeführt, wie man das Prinzip von Free Software für Nutzer, die Free Software bisher gemieden haben, zugänglich machen kann. Aus einem Treffen von führenden Experten im Jahr 1997 ging dann der Begriff *Open Source* hervor und es wurde ein Katalog an Anforderungen entworfen, welcher erfüllt sein musste, damit man von *Open Source Software* sprechen konnte. Die grundlegende Arbeit für diesen Katalog hatte Bruce Perens in dem Dokument "Debian Social Contract" umgesetzt. Das Dokument entstand, als Perens das Debian Projekt geleitet hatte. Dieses Projekt ist eine Distribution von Linux, welche nur Software erlaubt, die im Sinne der GNU Definition ist. "Debian Social Contract" beschreibt dabei, was die Software erfüllen muss, um GNU zu erfüllen. Die *Open Source Definition* (OSD) ist ein direkter Nachfolger dieses Dokuments und ist deshalb auch im Sinne der GNU Definition. Ein wichtiger Punkt dabei ist, dass die OSD, welche in Anhang A zu finden ist, mehr Freiheiten im Zusammenspiel von Open Source Software und proprietärer Software lässt. (DiBona, Ockman & Stone, 1999, S. 9)
Um die oben genannten Interessen durchzusetzen und die OSD und die darauf basierenden Lizenz zu verwalten, wurde die *Open Source Initiative* (OSI)[1] gegründet.
Aktuell gibt es mehr als 70 verschiedene Open Source Lizenzen, die von der OSI anerkannt sind. Open Source wird immer verbreiteter und auch immer mehr Firmen verwenden Open Source Software oder stellen ihre Software unter Open Source Lizenzen zur Verfügung. Ein Grund hierfür ist, dass Open Source Software die Kosten stark reduzieren kann. Es gibt mittlerweile aber auch viele Entwickler und Firmen die der Meinung sind, dass Open Source Software sicherer ist als proprietäre Software. Bei einer Umfrage des *Future of Open Source Survey* (*The Future of Open Source*, 2014) haben 72% der Befragten angegeben, dass sie Open Source aufgrund der besseren Sicherheit nutzen. Des Weiteren wird angenommen, dass über 50% der Unternehmen zu Open Source beitragen und es anwenden werden. Eine Aussicht auf die nächsten zwei bis drei Jahre zeigt, dass die Bereiche Regierung zu 67%, der Bereich Bildung sogar zu 76% und die Medizin zu 45% von Open Source beeinflusst werden.

[1] http://opensource.org/

2.1.2 Lizenzen

Das Grundkonzept von Open Source Software ist, wie oben erläutert, immer gleich, jedoch wurde es durch eine Vielzahl von verschiedenen Open Source Lizenzen rechtlich festgehalten.
In diesem Abschnitt sollen die rechtlichen Grundlagen erklärt werden, welche eine Lizenz und im speziellen eine Open Source Lizenz ausmachen. Des Weiteren gibt es verschiedene Typen an Open Source Lizenzen, welche unterschieden werden können. Diese variieren hauptsächlich darin, wie einschränkend sie bei der Verbreitung des Source Codes sind.
Laut (Alspaugh, Asuncion & Scacchi, 2009b) können Software Lizenzen im Allgemeinen in vier Gruppen eingeteilt werden. Zunächst kann man unterscheiden, ob es sich um eine Open Source Lizenz oder eine proprietäre Lizenz handelt. Bei proprietären Lizenzen entfallen die typischen Rechte, welche eine Open Source Lizenz gewährt. Hierzu zählen beispielsweise das Recht, die Software zu modifizieren, den Source Code weiterzugeben und die Arbeit zu kopieren. Die Proprietären erlauben es meist, nur die Software zu nutzen und verbieten oft die weiteren Rechte.
Bei den Open Source Lizenzen gibt es jedoch einige Unterschiede. Hier kann man prinzipiell drei verschiedene Arten unterscheiden.

- **tolerant** (permissive): Diese Lizenzen haben nur sehr wenige Pflichten und gewähren sehr viele Rechte.
- **reziprok** (reciprocal): Diese Lizenzen gewähren viele Rechte, aber dafür gibt es spezielle Pflichten bei Derivaten.
- **propagierend** (propagating): Diese Lizenzen gewähren viele Rechte, aber dafür gibt es spezielle Pflichten auf "benachbarte" Arbeit.

Die speziellen Pflichten bei reziproken Lizenzen sind meistens, dass der Source Code des Derivats unter der gleichen Lizenz veröffentlicht werden muss wie der originale Source Code.
Propagierende Lizenzen gehen in diesem Punkt noch einen Schritt weiter und verlangen, dass der Source Code von "benachbarter" Arbeit unter der gleichen Lizenz veröffentlicht werden muss wie der originale Source Code. Die Definition von "benachbart" kann dabei je nach propagierender Lizenz variieren und muss deshalb für jeden einzelnen Fall separat betrachtet werden.
Open Source Lizenzen basieren auf soliden gesetzlichen Grundlagen. Zu diesen Grundlagen zählen das Urheberrecht, das Patentrecht und das Vertragsrecht.
Das Urheberrecht gehört genau wie das Patentrecht zum geistigen Eigentumsrecht. Da Software geistiges Eigentum ist, gelten dafür genauso, wie für jedes andere geistige Eigentum, die entsprechenden Rechte.

Das Urheberrecht als eines dieser Rechte wird immer automatisch an ein schriftlich, akustisch oder auf eine andere materielle Art und Weise erstelltes Dokument angehängt. Dadurch ist diese Arbeit für eine lange Zeit geschützt. In Deutschland sind es 70 Jahre nach dem Tod des Urhebers[2]. Ist der Urheber nicht bekannt, so gilt diese Zeitspanne nicht erst ab dem Tod des Verfassers, sondern ab dem Zeitpunkt, ab dem die Arbeit veröffentlicht wurde[3]. Im amerikanischen Recht verhält sich dies wiederum anders. Hier erlischt das Urheberrecht bei unbekanntem Verfasser erst 95 Jahre nach der Veröffentlichung oder 120 Jahre nach der Erstellung, je nachdem welcher Zeitpunkt früher eintritt. Es ist nicht nötig, für ein Urheberrecht einen Antrag oder Ähnliches zu stellen - es ist ab dem Zeitpunkt der Anfertigung rechtlich bindend. (Laurent, 2004, S. 1-2)
Durch das Urheberrecht werden dem Urheber einige exklusive Rechte zugesichert über die er frei verfügen kann.
Zu diesen Rechten gehört:

- "You have an exclusive right to make copies.
- You have an exclusive right to prepare derivative works.
- You have an exclusive right to distribute copies of the original work or derivative works.
- In the case of certain kinds of works, including literary, musical, and motion picture works, you have an exclusive right to perform the work publicly.
- In the case of certain kinds of works, including literary, musical, pictorial, and sculptural works, you have an exclusive right to display the work publicly."(Rosen, 2004, S. 22-23)

Es gibt jedoch zwei wichtige Punkte, die beim Urheberrecht beachtet werden müssen. Dazu gehören die Prinzipien *work for hire* und *fair use*. Bei work for hire geht es darum, dass ein Angestellter für seinen Arbeitgeber Arbeit erstellt. Auch diese Arbeit ist urheberrechtlich geschützt, aber dieses Urheberrecht gehört dem Arbeitgeber, da er den Arbeitnehmer für die Arbeit bezahlt. Bei fair use geht es dagegen darum, unter welchen Umständen auch ein Anderer als der Urheber die Arbeit nutzen darf. Hierzu zählt z.B. wenn die Arbeit kommentiert wird oder wenn sie für Unterrichtszwecke verwendet wird. Ob die Verwendung einer Arbeit als fair use angesehen wird oder nicht, hängt auch stark davon ab, wie sehr es dem Urheber erschwert wird, seine Arbeit kommerziell zu vermarkten. Ist diese Beeinträchtigung zu stark, handelt es sich nicht mehr um fair use. (Laurent, 2004, S. 2)

[2]http://dejure.org/gesetze/UrhG/64.html
[3]http://dejure.org/gesetze/UrhG/66.html

Das Patentrecht ist ein weiteres Recht im geistigen Eigentum. Es zielt aber nicht auf die erstellte Arbeit selbst ab, sondern auf die zugrundeliegende Idee, welche als Patent angemeldet werden kann. Dies ist auch einer der Unterschiede zum Urheberrecht. Beim Patentrecht muss ein Antrag bei der entsprechenden Behörde, in Deutschland das Deutsche Patent- und Markenamt (DPMA)[4], gestellt werden, welcher einen längeren Prozess nach sich zieht. In diesem Prozess wird die Idee, welche als Patent angemeldet werden soll, auf Neuheit, Nichtoffensichtlichkeit und weitere rechtliche Tests geprüft. Wenn diese erfolgreich sind, wird die Arbeit als Patent anerkannt. (Rosen, 2004, S. 18)
Solch ein Patent gewährt dem Urheber wiederum bestimmte exklusive Rechte. Zu diesen zählen:

- "You have a right to exclude others from making products embodying your patented invention.
- You have a right to exclude others from using products embodying your patented invention.
- You have a right to exclude others from selling or offering for sale products embodying your patented invention.
- You have a right to exclude others from importing products embodying your patented invention."(Rosen, 2004, S. 23)

Wie man erkennen kann, handelt es sich um Rechte, welche anderen Personen Dinge verbieten.

Neben den Bestimmungen des geistigen Eigentumsrecht muss bei Lizenzen auch immer unterschieden werden, ob es sich um eine kombinierte Arbeit oder um ein Derivat handelt. Die beiden Begriffe stammen aus dem amerikanischen Recht und ihre Definitionen dort lauten:

> *"A "collective work" is a work, such as a periodical issue, anthology, or encyclopedia, in which a number of contributions, constituting separate and independent works in themselves, are assembled into a collective whole."* (17 U.S.C. § 101, 2015)

Das bedeutet, dass eine kombinierte Arbeit eine Arbeit ist, welche aus mehreren Softwarepaketen besteht, die zu einem Ganzen zusammengefügt und als solches auch weiterverbreitet wird.

[4]http://www.dpma.de/

> *"A "derivative work" is a work based upon one or more preexisting works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which a work may be recast, transformed, or adapted. A work consisting of editorial revisions, annotations, elaborations, or other modifications which, as a whole, represent an original work of authorship, is a "derivative work"."* (17 U.S.C. § 101, 2015)

Ein Derivat ist somit eine Arbeit, welche auf Basis einer anderen entsteht, beispielsweise durch Umgestaltung oder Anpassung. In dieser Definition ist jedoch nicht genau festgelegt, was eine Umgestaltung, eine Transformation oder eine Anpassung bedeutet. Dies muss im Zweifel ein Gericht entscheiden. (Rosen, 2004, S. 26-28)

2.1.3 FOSSology

FOSSology ist ein Projekt das von Hewlett-Packard ins Leben gerufen wurde. Es hat als internes Projekt zur Analyse von FOSS begonnen. Das Hauptaugenmerk liegt darauf, Dateien zu analysieren und deren Open Source Lizenzen zu bestimmen.
FOSSology selbst hat eine Webschnittstelle, über welche Dateien für die Analyse hochgeladen werden können und worüber die Analyse der Dateien gesteuert werden kann. Die Ergebnisse der Analyse werden anschließend in einer Datenbank gespeichert, sodass der Nutzer permanenten Zugriff darauf hat. Die Analysekomponente von FOSSology besteht aus vielen verschiedenen Agenten, welche je nach Art der Anwendung in unterschiedlicher Reihenfolge nacheinander ausgeführt werden.
Besonders interessant ist hierbei der *nomos* Agent. Dieser Agent ist dafür zuständig, für eine gegebene Datei oder ein gegebenes Archiv die vorhandenen Lizenzen zu extrahieren. Das Resultat dieser Extraktion ist eine Liste mit dem Dateinamen und der gefundenen Lizenz oder für das Archiv eine Auflistung aller enthaltenen Dateien und ihre zugehörigen Lizenzen. Nomos kann entweder über die Webschnittstelle von FOSSology verwendet werden oder über eine Kommandozeile aufgerufen werden.
Um Lizenzen bestimmen zu können, verwendet FOSSology einen Algorithmus zur Mustererkennung. Dieser Algorithmus wird *Symbolic Alignment Matrix (bSAM)* genannt und verwendet den *Protein Alignment Matrix* Algorithmus von Dayhoff. bSAM wird in FOSSology für mehrere Analysen genutzt. Bei der Lizenzanalyse wird er beispielsweise in Kombination mit weiteren Agenten verwendet, um die richtige Lizenz zu extrahieren. Dabei wird die Datei zunächst von einem *Word tokenizing filter* analysiert und anschließend versucht bSAM auf Basis dieser To-

kens durch die Mustererkennung die richtige Lizenz zu bestimmen. (Gobeille, 2008)

2.2 Verwandte Arbeiten

2.2.1 Vergleich von Lizenzen

Auf dem Gebiet der Lizenzanalyse gibt es frühere Arbeiten, welche sich damit beschäftigen, wie Lizenzen miteinander verglichen werden können und wie sich die Art der Verbindung darauf auswirkt. Die wichtigsten Arbeiten werden im Folgenden vorgestellt.

Arbeiten von Daniel German
Zunächst wird eine Arbeit von German vorgestellt (German & Hassan, 2009), welche sich damit beschäftigt, wie Lizenzen modelliert werden können, so dass sie maschinenlesbar und -interpretierbar sind und wie sich bestimmte Lizenzen auf die Wiederverwendung von Komponenten auswirken. Ein weiterer Punkt, der diskutiert wird, ist die Kompatibilität von Lizenzen in einem Komponentenbasierten Softwaresystem, basierend auf dem vorher definierten Modell. Das Hauptaugenmerk des Papers liegt jedoch auf der Identifikation von Patterns für die Integration von Komponenten mit unterschiedlichen Lizenzen in ein Komponentenbasiertes Softwaresystem.
Für diese Arbeit sind aber nur das entwickelte Modell des Softwaresystems, der Lizenz und die Kompatibilität von Lizenzen wichtig. Diese Punkte werden im Folgenden näher erörtert.
Ein Komponentenbasiertes Softwaresystem besteht aus mehreren Komponenten, welche alle eine eigene Lizenz haben können. Für die Betrachtung des Zusammenspiels unterschiedlicher Lizenzen ist es nicht nur wichtig zu wissen, um welche Lizenzen es sich handelt, sondern auch wie die Komponenten, für welche die Lizenzen gelten, im Gesamtsystem wiederverwendet werden. Dies ist notwendig, um feststellen zu können, ob es sich bei der Wiederverwendung um ein Derivat des Originalwerks handelt oder um eine kollektive Arbeit. Für die Wiederverwendung werden in dem Paper zwei Arten definiert:

- **Whitebox**: Whitebox bedeutet, dass ein oder mehrere Dateien der Komponente in ihrem Originalzustand oder in modifizierter Form wiederverwendet werden. Dies führt in nahezu jeder Form zu einem Derivat.
- **Blackbox**: Blackbox bedeutet, dass die Komponente ummodifiziert als Teil des Gesamtsystem oder separat verwendet wird.

Für die sogenannte Blackbox Wiederverwendung wurden nun zusätzlich noch fünf Verbindungsarten definiert, welche unterschieden werden müssen, sofern bestimmt werden soll, ob es sich um ein Derivat oder eine kollektive Arbeit handelt. Die definierten Verbindungstypen sind:

- Linking.
- Fork.
- Subclass.
- Inter Process Communication (IPC).
- Plugin.

Anschließend kann nun das Modell einer Lizenz definiert werden.
Eine Lizenz besteht aus einem oder mehreren Rechten, für die wiederum keine oder mehrere Pflichten definiert werden. Die Pflichten werden als eine Menge von Konjunktionen definiert. Um ein Recht zu erhalten, müssen alle Konjunktionen erfüllt sein.
Um nun eine solche Lizenz in das Gesamtsystem integrieren zu können, muss der Integrator zwei Faktoren beachten:

- Er muss beachten, ob das System ein Derivat der Komponente ist. Dies wiederum ist abhängig von den oben beschriebenen Verbindungstypen.
- Des Weiteren muss der Integrator das Recht haben, die Komponente zu nutzen und gegebenenfalls auch weiterzuverbreiten.

Nachdem nun das Modell einer Lizenz und ihr Zusammenspiel definiert wurde, werden im Folgenden die definierten Regeln für die Kompatibilität von Lizenzen aufgezeigt:

- Jedes Recht des Systems S benötigt eine Genehmigung zu einem oder mehrerer Rechte von jeder Komponente *C1, ... , Cn.*
- Jede Genehmigung hat eine Menge an Konditionen, welche als Konjunktionen modelliert sind. Wenn die Vereinigung aller Konjunktionen nicht erfüllbar ist, kann S nicht die gewählte Lizenz haben.
- Für eine gegebene Genehmigung g, ist *L(c)* nicht kompatibel mit *L(s)* wenn mindestens eine der Konjunktionen von g von *L(c)* nicht erfüllbar ist unter *L(s).*
- Die Nutzung von C unter S ist erlaubt, wenn die Nutzung durch eine Genehmigung gewährt wird und diese Genehmigung auch erfüllt ist.

In dieser Publikation wurden zwar einige Definitionen für ein Lizenzmodell vorgestellt, sowie Regeln für das Zusammenspiel und für die Kompatibilität von Lizenzen. Es gibt jedoch auch einige Schwachpunkte, welche eine Integration

des Ganzen in ein automatisiertes Softwaresystem erschwert. Bei der Modellierung einer Lizenz fehlt die Beschreibung, welche Rechte wie im Detail modelliert werden. Des Weiteren fehlt eine Beschreibung darüber, wie die Konjunktionen miteinander verglichen werden sollen. Hierfür müssten Regeln definiert werden, die festlegen, wann zwei Rechte oder zwei Pflichten miteinander kompatibel sind.

Arbeiten von Walt Scacchi
Weitere wichtige und interessante Arbeiten auf dem Gebiet der Lizenzanalyse und im speziellen der Analyse von Konflikten zwischen verschiedenen Lizenzen sind (Alspaugh, Scacchi & Asuncion, 2010) (Alspaugh et al., 2009b) (Alspaugh et al., 2009a).
Diese Arbeiten von Alspaugh und Scacchi beschäftigen sich alle mit der Entwicklung eines Meta-Modells für Lizenzen und damit wie für ein Softwaresystem, bestehend aus vielen unterschiedlich lizenzierten Komponenten, die Rechte (rights) und Pflichten (obligations) für das Gesamtsystem berechnet werden können.
Es wird dabei zunächst die Analyse von bestehenden Open Source Lizenzen mit Hilfe eines Ansatzes basierend auf der "semantischen Parametrisierung" (Breaux, Antón & Doyle, 2008) durchgeführt.
Anschließend wird eine Lizenz als eine Menge von Rechten dargestellt. Diese Rechte besitzen wiederum keine oder mehrere Pflichten. Die Struktur für Rechte und Pflichten ist jeweils die gleiche - sie bestehen aus einem Tupel (Actor, modality, action, object).
Um die Rechte und Pflichten für ein bestimmtes System zu bestimmen, wird über die Lizenzen des Systems iteriert und die jeweiligen Lizenzen mit den dazugehörigen Komponenteninformationen instanziiert. Aus dem daraus resultierenden konkreten Rechten und Pflichten können diejenigen bestimmt werden, die für das Gesamtsystem gelten, oder mögliche Konflikte erkannt werden.
Im Folgenden werden die hier nur grob erläuterten Ansätze im Detail erörtert.

Analyse von Software Lizenzen
Die Analyse der Lizenzen basiert auf dem Prinzip der "semantischen Parametrisierung". Sie umfasst zunächst 3 Schritte:

1. Es wird zuerst eine Konkordanz gebildet, welche die wichtigsten Wörter der Lizenzen indiziert und die Sätze mit ihrer zugehörigen Sektion, Paragraph und Satznummer annotiert. Füllwörter, Artikel etc. mit keiner bestimmten Bedeutung für die Lizenz wurden dabei vernachlässigt.

2. Irrelevante Teile der Lizenzen wie Einleitung etc. wurden entfernt.

3. Mit den Konkordanzen von allen untersuchten Lizenzen wurden nun Wörter und Sätze mit derselben Bedeutung identifiziert. Danach wurden natürliche Sprachmuster identifiziert, welche als *Restricted Natural Language Statement* (RNLS) verwendet werden können, um eine Lizenz darzustellen.

Aus diesen RNLS wurde anschließend folgendes Meta-Modell abgeleitet:

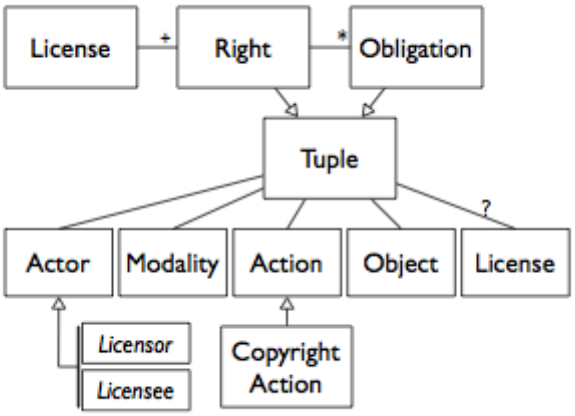


Abbildung 2.1: Lizenz Meta-Model aus (Alspaugh et al., 2009a)

Wie man erkennt, besteht eine Lizenz (License) aus mehreren Rechten (Right), welche wiederum keine oder mehrere Pflichten (Obligation) haben. Die Rechte und Pflichten bestehen aus Tupel (Tuple). Diese beinhalten einen Aktor (Actor), eine Modalität (Modality), eine Aktion (Action), ein Objekt (Object) für die Aktion und optional eine Lizenz, auf welche sich die Aktion bezieht.

Die möglichen Modalitäten, Objekte und Lizenzen sind folgendermaßen definiert:

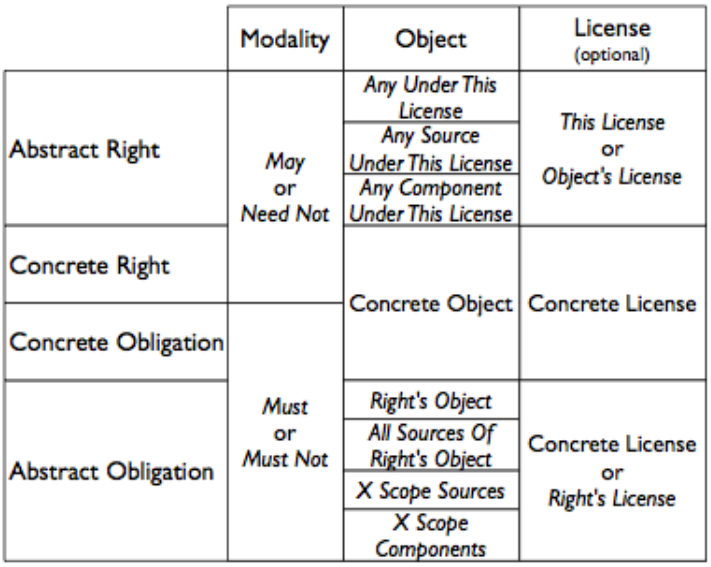


Abbildung 2.2: Modalitäten, Objekte und Lizenzen für abstrakte und konkrete Rechte und Pflichten aus (Alspaugh et al., 2009a)

Man kann dem Diagramm entnehmen, dass die Modalitäten “may” und “need not” zu den Rechten gehören und dass die Modalitäten “must” und “must not” zu den Pflichten gehören. Die abstrakten Objekte sind die Platzhalter, bis sie bei der Instanziierung durch konkrete Objekte ersetzt werden.
Bei den Aktionen kann unterschieden werden zwischen denen, die sich auf das Urheberrecht beziehen und die Aktionen, welche aus den RNLS abgeleitet wurden.
Die RNLS Aktionen sind mit Tokens definiert, welche anzeigen, an welcher Stelle die Objekte des Tupels und, falls vorhanden, die Lizenz eingefügt wird.
Nach meinem Verständnis stellen diese RNLS Aktionen abstrakte Rechte/Pflichten dar und sobald die Objekte instanziiert werden, werden sie zu konkreten Rechten/Pflichten.
Die Copyright Aktionen bilden eine partielle Ordnung, in der die höher geordneten Aktionen die Aktionen unter ihnen, mit denen sie verbunden sind, voraussetzen. Die Aktionen aus den RNLS werden darin mit integriert, um die vollständige Ordnung der Aktionen zu erhalten.
Genauere Beschreibungen hierzu sind leider in keinem der Paper vorhanden. Um dies selbst nachzubilden wäre eine detailliertere Beschreibung sehr wichtig.
Die weiteren Informationen, die man nun noch zu dem System benötigt, sind durch die Liste an Quantifikatoren, welche als Objekte in den Rechten und Pflichten auftreten können, definiert.

Die Information, welche dabei benötigt wird, wird Lizenzarchitektur genannt und setzt sich aus folgenden Punkten zusammen:

- Die Komponenten des Systems.
- Die Beziehung, die jeder Komponente ihre Lizenz zuordnet.
- Die Beziehung, die jeder Komponente ihre Sources zuordnet.
- Die Beziehung jeder Komponente zu der Menge an Komponenten im gleichen Lizenzraum. Dies gilt für jede Komponente, in der ein Lizenzraum definiert ist.

Analyse der Lizenzarchitektur
Mit dem oben beschriebenen Lizenzmodell und der Beschreibung, welche Informationen für das System notwendig sind, kann nun das Gesamtsystem untersucht werden.
Es wird zunächst über die Lizenzen der Komponenten iteriert, um diese mit den nötigen Komponenteninformationen zu versorgen.
Dieser Vorgang sieht folgendermaßen aus:
Jedes der abstrakten Rechte in jeder Lizenz hat als ihr Objekt entweder:

- "Any Under this License"
- oder "Any Source Under this License"
- oder "Any Component Under this License"

entsprechend der Abbildung 2.2.

Ein abstraktes Recht in einer Lizenz wird folgendermaßen in ein oder mehrere konkrete Rechte umgewandelt:

- "Any Component" wird nacheinander mit jeder Komponente, welche die Lizenz hat, ersetzt.
- "Any Sources" wird nacheinander mit jeder Source, welche die Lizenz hat, ersetzt.
- "Any" wird mit beiden ersetzt. (Komponente und Source).
- Hat das abstrakte Recht "Objects License" als Lizenz, so wird in jedem konkreten Recht die Lizenz mit der jeweiligen Lizenz des Objektes ersetzt (Objekte können Referenzen sein, deshalb gibt es hier die Unterscheidung zwischen "This License" und "Objects License").

Anschließend müssen die zugehörigen abstrakten Pflichten in konkrete Pflichten umgewandelt werden. Dies geschieht folgendermaßen:

- Hat die Pflicht als Objekt "Rights Object", so gibt es nur eine Pflicht, welche als Objekt das Objekt des zugehörigen konkreten Rechtes hat.
- Hat die Pflicht als Objekt "All Sources of Rights Object", dann gibt es eine Pflicht für jede Source des Objektes des konkreten Rechtes (das Objekt des konkreten Rechts muss in diesem Fall eine Komponente sein). Das Objekt für jede Pflicht ist die jeweilige Source.
- Hat die Pflicht als Objekt "X Scope Components" (X stellt dabei eine Lizenz dar), dann gibt es eine Pflicht für jede Komponente, welche im Gültigkeitsbereich des Objektes des konkreten Rechts liegt. Als Definition für den Gültigkeitsbereich gilt dabei die, die in X definiert ist. Die konkreten Pflichten erhalten dann als konkretes Objekt die Komponente.
- Hat die Pflicht als Objekt "X Scope Sources" (X stellt dabei eine Lizenz dar), dann gibt es eine Pflicht für jede Source der Komponenten, welche im Gültigkeitsbereich des Objektes des konkreten Rechtes liegen. Als Definition für den Gültigkeitsbereich gilt dabei die, die in X definiert ist. Die konkreten Pflichten erhalten dann als konkretes Objekt die Source.

Nun hat man eine Liste an Lizenzen mit den auf obige Art instanziierten konkreten Rechten und Pflichten. Anschließend muss festgelegt werden, wie diese konkreten Pflichten erfüllt werden können.

- Zunächst wird das korrelierte Recht gebildet. Das geschieht, indem aus den Modalitäten die jeweils korrelierende gewählt wird ("may" für "must", "need not" für "must not") und die restlichen Teile der Pflicht hinzugenommen werden.
- Ist das korrelierte Recht ein Copyright Recht, so muss die Lizenz des Objektes genutzt werden, um es zu erfüllen. Da das Objekt aus der Pflicht mit übernommen wurde, bedeutet dies, dass die Lizenz des Objekts der konkreten Pflicht verwendet wird. Es wird die Lizenz gesucht und innerhalb der Lizenz ein abstraktes Recht, welches das korrelierte Recht generalisiert.
- Für das abstrakte Recht und das korrelierte Recht wird dann der Prozess wiederholt. Hierbei stellt sich die Frage, welcher Prozess damit gemeint ist.
- Gibt es das abstrakte Recht, welches das korrelierte Recht generalisiert, nicht, wird nach Nachfolgern für das Copyright Recht in der partiellen Ordnung aller Copyright Rechte gesucht und versucht, diese zu erfüllen. Gelingt das auch nicht, wird das korrelierte Recht als unerfülltes korreliertes Recht gespeichert.

- Ist das korrelierte Recht kein Copyright Recht, muss es nicht über eine Lizenz bezogen werden. Es muss aber trotzdem überprüft werden, ob es in Konflikt mit einer anderen Pflicht steht. Dafür muss das korrelierte Recht am Ende, abschließend mit allen Pflichten verglichen werden. Gibt es eine Pflicht, die genauso aufgebaut ist wie das Recht nur mit "umgekehrter" Modalität, so stellt dies ein *Right-Obligation-Conflict* dar.
- Zum Schluss werden alle Pflichten durchlaufen auf der Suche nach Paaren, bei denen sich nur die Modalitäten unterscheiden. In dem Fall liegt dann ein *obligation - obligation conflict* vor.

Trotz dieser Beschreibung ist unklar, wie ein Vergleich von Rechte und Pflichten im Detail aussieht. Beispielsweise ist nicht wirklich klar, wann eine Pflicht erfüllt ist. Eine Annahme wäre, dass die Pflicht erfüllt ist, wenn das korrelierte Recht und das Recht, zu dem die Pflicht gehört, übereinstimmen.

Licenses Compatibility and Composition in the Web of Data
Die letzte Arbeit zu dem Thema Vergleich von Lizenzen stammt von (Villata & Gandon, 2012) und beschäftigt sich nicht direkt mit Open Source Lizenzen. Sie ist dennoch interessant, da sie einen ähnlichen Ansatz verfolgt, wie der in dieser Arbeit gewählte. Besonders hervorzuheben ist hierbei, dass der Ansatz aus dieser Arbeit unabhängig von diesem Paper entstanden ist und ich auf dieses Paper erst im Anschluss daran gestoßen bin.
Eine der Fragen, welche das Paper klären will, ist die Kompatibilität von Lizenzen. Dafür verwendet der Autor genauso wie der Ansatz aus dieser Arbeit das *Creative Commons Rights Expression Language* (ccREL) Schema. Für den Test auf Kompatibilität werden für die einzelnen Elemente aus der Sprache Regeln definiert. Diese Regeln legen fest, wie die verschiedenen Rechte und Pflichten, welche mit ccREL definiert wurden, miteinander verglichen werden können.
Da diese Arbeit sich nicht direkt mit Softwarelizenzen beschäftigt, werden hier auch keine Regeln definiert, welche die Kompatibilität von Lizenzen, bezogen auf deren Verbindungstyp, festlegen. Ein weiterer Schwachpunkt für den Vergleich ist das geringe Vokabular von ccREL. Damit ist es schwierig, eine Vielzahl von Lizenzen ausreichend detailliert genug abzubilden.

2.2.2 Analyse von Softwarearchitekturen

In diesem Abschnitt sollen Arbeiten vorgestellt werden, welche sich mit der Analyse von Softwarearchitekturen beschäftigen. Zum einen geht es um eine Arbeit, welche die Wiederherstellung einer Softwarearchitektur aus dem Objektorientierten Code beschreibt, und um eine Arbeit, welche für die Komponenten einer

Architektur, für die nur ein *Jar-Archiv* zur Verfügung steht, die Lizenzeigenschaften identifiziert. Zum Anderen wird eine Arbeit vorgestellt, welche sich damit beschäftigt, die gegenseitige Abhängigkeit von Software zur Übersetzungs- und Laufzeit zu verstehen.
Diese Arbeiten werden hier vorgestellt, da sie interessante Ansätze für die Generierung von Softwarearchitekturen und die Identifikation von Lizenzen darstellen. Die Ansätze sind im Besonderen interessant für den weiteren Ausbau der in dieser Arbeit vorgestellten Anwendung.
Möchte man für eine Softwareanwendung die Architektur analysieren, ist es zunächst einmal wichtig, diese vollständig dokumentiert zu haben. Ist dies nicht der Fall oder wurde im Vorfeld der Entwicklungsarbeit gar keine Designphase durchgeführt, muss die Architektur aus dem bereits bestehenden Code gewonnen werden.
Ein Ansatz für diese Wiederherstellung ist im Paper "A Genetic Approach for Software Architecture Recovery from Object-Oriented Code" (Seriai & Chardigny, 2011) beschrieben. Es wird ein genetischer Algorithmus entwickelt, welcher auf Basis einer Fitnessfunktion aus einer Anfangsmenge an Lösungen versucht, immer neue und bessere Lösungen zu finden. Inspiriert sind diese Algorithmen durch die Evolutionsmechanismen, bekannt aus der Biologie. Für die Fitnessfunktion werden mehrere Eigenschaften definiert, welche messen, wie unabhängig eine Komponente ist, wie groß die Kombinierbarkeit der Komponenten ist und wie viel Funktionalität in den Komponenten steckt. Diese Eigenschaften werden anschließend in Metriken umgewandelt und es werden basierend darauf Formeln entwickelt, welche in linearer Kombination zu einer Auswertungsfunktion der semantischen Korrektheit führen. Es wird gezeigt, dass mit Hilfe dieses Ansatzes für eine Beispielarchitektur ein Wert von 80,2% für die Fitnessfunktion erreicht werden kann. Des Weiteren wird aufgezeigt, dass dieser Ansatz ein besseres Verhältnis zwischen Ausführungszeit und Qualität der Ergebnisse liefert, als vergleichbare Ansätze, welche Clustering Algorithmen verwenden.
Sobald man eine Architektur zur Verfügung hat, unabhängig davon, ob diese automatisch berechnet wurde oder über eine Dokumentation erstellt wurde, müssen die Abhängigkeiten zwischen den Komponenten bestimmt werden. Ohne diese Abhängigkeiten ist beispielsweise keine Analyse auf Lizenzkonflikte möglich. Ein Ansatz, um die Abhängigkeiten zu bestimmen, wird in dem Paper "A Model to Understand the Building and Running Inter-Dependencies of Software" (Germán et al., 2007) beschrieben. Zunächst wird in dem Paper festgelegt, wie die Abhängigkeiten zwischen Paketen klassifiziert werden können. Die Kriterien dafür sind: *Type*, *Importance*, *Stage* und *Usage Method*. Eine Übersicht darüber ist in folgender Tabelle zu sehen:

Classification	Examples
Type	Explicit Abstract
Importance	Required Optional
Stage	Build Installation Run-Time Testing
Usage method	Stand-alone Middleware-based Plug-in Linkable library

Tabelle 2.1: Klassifikation der Abhängigkeiten nach (Germán et al., 2007)

Um diese Abhängigkeiten zu erhalten, wird vorgeschlagen, sich die Dateien des *Package Management Systems* anzusehen und aus diesen die Abhängigkeiten zu extrahieren. Anschließend werden die extrahierten Abhängigkeiten mittels eines Abhängigkeitsgraphen dargestellt. Dies ist eine gute Methode, um die Abhängigkeiten graphisch darzustellen. Jedoch besteht das Problem, dass die Abhängigkeiten nur schwer identifiziert werden können, sofern kein Package Management System verwendet wurde.
Um eine Softwarearchitektur auf deren Lizenzkonflikte zu untersuchen, werden die Lizenzen für die einzelnen Komponenten der Architektur benötigt. Liegt für eine Komponente jedoch nur eine *Jar* Datei vor, ist es schwierig, dessen Lizenz zu bestimmen. Ein Ansatz, um dieses Problem zu lösen, ist in "Identifying Licensing of Jar Archives using a Code-Search Approach" (Penta, Germán & Antoniol, 2010) beschrieben. Die Grundidee dieses Papers ist die Lizenz eines Jar-Archives zu bestimmen, indem die Nutzung von *Code Search Engines* und die automatische Klassifikation der Lizenzen, welche in Textdateien innerhalb des Jars gefunden werden, kombiniert werden. Zunächst werden die Textdateien aus den Jar-Archiven extrahiert und anschließend innerhalb dieser Textdateien nach Lizenzen gesucht. Die gefundenen Lizenzen werden dann klassifiziert. Der nächste Schritt ist es, für die enthaltenen Java Dateien mittels der *Google Code Search Engine* Informationen abzufragen. Dafür muss aus der *.Class* Datei der Paket- und Klassenname bestimmt werden. Mit Hilfe dieser Namen wird dann in der *Google Code Search Engine* nach der Datei gesucht. Bei erfolgreicher Suche erhält man eine XML-Ausgabe mit Informationen, welche unter anderem die Lizenz enthält. Mit dieser Methode erzielten die Autoren in einer von ihnen durchgeführten Studie eine Genauigkeit von 95%. Somit ist dieser Ansatz sehr gut geeignet, um Lizenzen von Jar Dateien zu bestimmen. Nachteil dieses Verfahrens ist jedoch, dass es nur auf Jar-Archive anwendbar ist und dass die *Google Code Search Engine*

nicht mehr betrieben wird. Somit müsste man diese durch eine andere Suchmaschine für Source Code ersetzen.

2.3 Konzept und Modell

Dieser Abschnitt beschreibt das Konzept und die im Rahmen der Arbeit entwickelten Modelle. Zunächst werden die Anforderungen an die einzelnen Modelle zusammengefasst. Im Anschluss daran werden die Modelle vorgestellt und wichtige Entscheidungen bezüglich des Aufbaus genauer erläutert.

2.3.1 Anforderungen

Für das Modell einer Architektur, für die maschinenlesbare Darstellung einer Lizenz, für den Vergleich von Lizenzen und für den Editor zur Darstellung und Bearbeitung von Softwarearchitekturen gibt es einige Anforderungen. Diese werden im Folgenden genauer erläutert. Die Grundidee, welche dieser Arbeit zugrunde liegt, ist es, eine komponentenbasierte Softwarearchitektur mittels einer Anwendung graphisch darstellen, editieren und erweitern zu können. Auf Basis dieser Anwendung sollte es möglich sein, die Architektur mit Meta-Daten zu annotieren und Aussagen über die Architektur treffen zu können. Am wichtigsten ist dabei die Konfliktanalyse von unterschiedlich lizenzierten Komponenten der Architektur.

Darstellung der Architektur
Um überhaupt erst eine Architektur in einem Editor darstellen zu können, muss eine Repräsentation festgelegt werden. Es sollte möglich sein, eine Architektur unabhängig vom Editor festlegen und dann einlesen zu können. Dies erleichtert den Austausch von Architekturbeschreibungen. Aus diesem Grund wurde eine einfache *Architecture Description Language* (ADL) entwickelt, welche im Abschnitt 2.3.1 beschrieben wird.

Modell einer Lizenz
Möchte man Aussagen über die Kompatibilität von unterschiedlichen Open Source Lizenzen machen, braucht man eine Darstellung dieser Lizenzen in einer formalen, maschinenlesbaren Form. Wichtig dabei ist es einen Weg zu finden, die vielen verschiedenen Klauseln, welche durch die Vielzahl an verschiedenen Open Source Lizenzen zur Verfügung stehen, in eine formale Form zu überführen, mittels der man maschinelle Vergleiche durchführen kann. Ein wichtiger Schritt dabei ist es, die Punkte zu identifizieren, die bei allen Lizenzen gleich sind.

Lizenzvergleich
Nachdem nun Modelle für die Softwarearchitektur und für die Darstellung von Lizenzen erstellt wurden, muss auf Basis dieser Modelle eine Möglichkeit gefunden werden, die Architektur, mit Hilfe eines Algorithmus, auf mögliche Lizenzkonflikte zu untersuchen. Hierfür ist es notwendig, die einzelnen Lizenzen für jede Komponente in Erfahrung zu bringen. Des Weiteren ist es wichtig in Erfahrung zu bringen, welche Komponenten wie miteinander verbunden sind. Diese Information ist sehr wichtig um bestimmen zu können, inwiefern sich die Lizenzen der Komponenten gegenseitig beeinflussen. Manche Verbindungsarten stellen beispielsweise eine sogenannte *License-Firewall* (Alspaugh et al., 2009a) dar, welche verhindert, dass die Bestimmungen einer Lizenz für eine Komponente A auch für die über die "License-Firewall" verbundene Komponente B gelten.

2.3.2 Modell zur Beschreibung der Architektur

Um ein komponentenbasiertes Softwaresystem in einem Analyse-Werkzeug abbilden zu können, muss die Architektur des Systems in Form seiner Komponenten und den Verbindungen dargestellt werden. Hierfür und für die interne Verarbeitung wird ein Modell der Architektur benötigt. Für diesen Zweck wurden *Architecture Description Languages* entwickelt. Diese Sprachen stellen einen Spezialfall domänenspezifischer Sprachen dar. Sie sind dafür geeignet, Softwarearchitekturen zu modellieren, zu analysieren oder auch zu simulieren. Durch ihre formale Repräsentation sind sie eine präzise Darstellung einer Architektur, welche von einer Maschine und von Menschen lesbar ist. Ein weiterer Vorteil, den die Modellierung einer Architektur mittels einer formal definierten Sprache mit sich bringt, ist, dass die entworfenen Architekturen gut wiederverwendbar und unter Nutzern austauschbar sind. (Vogel et al., 2009, S. 279-283)
In dieser Arbeit wurde eine Architekturbeschreibung auf Basis von *xArch* (xArch, 2015) entworfen, um die oben beschriebenen Vorteile einer ADL nutzen zu können. xArch ist eine ADL, die für die Modellierung eines Softwaresystems eingesetzt wird. Entwickelt wurde diese ADL von der University of Californica, Irvine. Sie basiert auf einer einfachen XML Notation zur Modellierung von Softwarearchitekturen.
Die Beschreibung, welche in dieser Arbeit verwendet wird, besteht aus drei Hauptbestandteilen.

- **Component:** stellt eine Komponente der Softwarearchitektur dar. Sie hat folgende Eigenschaften:
  - Name.
  - Beschreibung.
  - Liste an Interfaces.

  - Art der Lizenz.
  - Lizenz.

- **Interface:** stellt eine Verbindungsmöglichkeit der Komponente dar. Die Verbindungsmöglichkeiten wurden hier als zusätzlicher Bestandteil modelliert, um klarer abzugrenzen, welche Komponente welche Eingänge oder Ausgänge besitzt. Sie enthält folgende Eigenschaften:
  - Beschreibung.
  - Richtung: hier kann angegeben werden, ob die Komponente, zu der das Interface gehört, eine ausgehende Verbindung, eine eingehende oder eine Kombination aus beidem akzeptiert.
- **Connection:** stellt eine Verbindung zwischen zwei Komponenten dar und enthält folgende Eigenschaften:
  - Beschreibung.
  - Quelle: ausgehendes Interface der Quellkomponente.
  - Ziel: eingehendes Interface der Zielkomponente.
  - Verbindungstyp.
  - Beziehungstyp.
  - Konflikt.
  - Konfliktbeschreibung.

Ergänzt werden die Hauptbestandteile durch festgeschriebene Werte für bestimmte Parameter. Hierzu zählen die erlaubten Lizenzen, die Art der Lizenz, die Verbindungstypen, die Richtung der Verbindung und ob ein Konflikt vorliegt. Die möglichen Werte für diese Parameter können folgender Tabelle entnommen werden:

Lizenzen	unknown, Apache-2.0, Artistic License, BSD-3-Clause, CTL, EPL, GPL-2.0, GPL-3.0, LGPL-2.1, LGPL-3.0, MIT, MS-PL
Art der Lizenz	Foss, Proprietary
Richtung der Verbindung	incoming, outgoing, inandout
Verbindungstyp	staticLink, dynamicLink, fork, ipc, linkedPlugin, forkPlugin, subclass
Konflikt	yes, no, unknown

Tabelle 2.2: Parameterwerte für ADL

Für die Arbeit wurden die erlaubten Lizenzen auf die zehn am häufigsten verwendeten Open Source Lizenzen[5] und eine Proprietäre Lizenz (Corel Transactional License (CTL))[6] beschränkt.
Ziel der Modellierung über ein separates Schema ist, dass der Nutzer sehr einfach eine Architektur modellieren kann und diese einfach editieren und im Besonderen auch mit Anderen teilen kann. Ein weiterer wichtiger Punkt, weshalb dieses Schema gewählt wurde, war, die einfache Art eine Architektur zu erstellen. Durch die Verwendung von bewährten Technologien wie XML kann diese Sprache schnell erlernt werden und bietet für viele Nutzer eine bereits bekannte Struktur.

2.3.3 Maschinenlesbare Darstellung einer Lizenz

Bei der Modellierung der Lizenzen wurden zwei verschiedene Modelle entwickelt. Diese werden im Folgenden vorgestellt.

Modell basierend auf ccREL
Die Darstellung der Lizenzen erfolgt bei diesem Modell mit Hilfe der ccREL. (Abelson, Adida, Linksvayer & Yergler, 2008) Es handelt sich dabei um eine *Rights Expression Language* (REL), welche verwendet wird, um deskriptive Metadaten an Medieninhalte anzuhängen. Die Sprache wurde jedoch auch weiterentwickelt, um mittlerweile auch weitere Lizenzen zu unterstützen und maschinenlesbar darzustellen.
Den Anfang dabei machte die *Free Software Foundation*, welche ihre Lizenzen, wie beispielsweise die GNU GPL, im ccREL Format veröffentlichten. (Linksvayer, 2009)
Die Sprache basiert auf dem *Ressource Description Framework* (RDF) und besteht aus einer kleinen Menge an *RDF* Eigenschaften. Diese beinhalten zwei Klassen an Eigenschaften, die *Work* Eigenschaften und die Lizenzeigenschaften. Für diese Arbeit sind die Lizenzeigenschaften das Interessante und deshalb werden diese im Folgenden genauer erläutert. Nach (*Describing Copyright in RDF*, 2015) wurden sechs verschiedene Lizenzeigenschaften definiert, welche wiederum Untereigenschaften enthalten können. Diese sechs Eigenschaften sind:

- **cc:permits:** erlaubt eine bestimmte Nutzung einer Arbeit im Rahmen des grundlegenden Copyrights und darüber hinaus.
- **cc:prohibits:** verbietet eine bestimmte Nutzung einer Arbeit, im Speziellen betreffend den Bereich der Genehmigungen aus *cc:permits* (es reduziert jedoch keine Rechte, welche durch das Urheberrecht gewährleistet sind).

[5] https://www.blackducksoftware.com/resources/data/top-20-open-source-licenses
[6] http://apps.corel.com/clp/terms.html

- **cc:requires:** legt bestimmte Aktionen fest, welche der Nutzer ausführen muss, wenn er die Genehmigungen aus *cc:permits* nutzen möchte.
- **cc:jurisdiction:** setzt die Lizenz mit einer bestimmten Rechtssprechung in Bezug.
- **cc:deprecatedOn:** weist darauf hin, dass die Lizenz ab dem gegebenen Datum veraltet (deprecated) ist.
- **cc:legalCode:** referenziert den Originaltext der Lizenz. Dies ist der Text, der rechtlich relevant ist.

Am wichtigsten von diesen sechs Eigenschaften sind *permits*, *prohibits* und *requires*, da diese speziell die Klauseln einer Lizenz betreffen. Für diese drei Eigenschaften werden im Folgenden noch die Untereigenschaften vorgestellt, da diese relevant für die Modellierung einer Lizenz mittels ccREL sind. *cc:permits* hat folgende Untereigenschaften:

- **cc:Reproduction:** beschreibt das Erstellen von mehreren Kopien des Originals.
- **cc:Distribution:** beschreibt die Weiterverbreitung der Arbeit.
- **cc:DerivativeWork:** beschreibt das Erstellen eines Derivats des Originals.
- **cc:Sharing:** erlaubt die Anfertigung von kommerziellen Derivaten, jedoch nur die nicht-kommerzielle Verteilung.

cc:prohibits hat folgende Untereigenschaften:

- **cc:CommercialUse:** verbietet die Nutzung der Arbeit für kommerzielle Zwecke.

cc:requires hat folgende Untereigenschaften:

- **cc:Notice:** bedeutet, dass Copyright und Lizenzhinweise intakt bleiben müssen.
- **cc:Attribution:** Copyright Inhaber und/oder Ersteller müssen erwähnt werden und "Anerkennung" bekommen.
- **cc:ShareAlike:** abgeleitete Arbeit muss unter der gleichen Lizenz weiterverbreitet werden.
- **cc:SourceCode:** wird die Arbeit weiterverbreitet, muss der Source Code mit veröffentlicht werden.
- **cc:NetworkSource:** der Source Code muss für ein Derivat und kombinierte Arbeit veröffentlicht werden, auch wenn diese nur über ein Netzwerk angeboten wird. Trifft im Speziellen auf die *GNU Affero General Public License* (AGPL) Lizenz zu.

- **cc:CopyLeft:** Derivate und kombinierte Arbeit muss unter den gleichen Bedingungen wie das Original lizensiert werden.
- **cc:LesserCopyLeft:** Ein Derivat muss unter denselben Bedingungen wie das Original lizensiert werden. Eine kombinierte Arbeit kann durch eine andere Lizenz lizenziert werden.

Mit Hilfe dieser Eigenschaften können nun Lizenzen formal modelliert werden. Ein Beispiel dafür ist die formalisierte Darstellung der GNU General Public License Version 3 (GPLv3)[7], welche sich in Anhang B befindet.
Für diese Arbeit wurde *cc:requires* um eine weitere Eigenschaft ergänzt. Da es mehrere wichtige Open Source Lizenzen, wie beispielsweise die GPLv3, die Apache License Version 2 etc., mit Patentklauseln gibt, wurde die Eigenschaft *masterThesis:Patent* eingeführt. Diese besagt, dass die Patente, welche für die Arbeit notwendig sind, mit lizenziert werden müssen.
Eine weitere Ergänzung des ccREL Schemas erfolgt durch die Erweiterung um die aus der Arbeit von German (Germán et al., 2007) bekannten Verbindungstypen. *Linking* wurde dabei noch einmal unterteilt in *static* und *dynamic linking* und *Plugin* in *linked Plugin* und *forked Plugin*. Die weitere Unterteilung bei *Plugin* resultiert daraus, dass ein Plugin, welches mittels eines linking Mechanismus eingebunden wurde, bei bestimmten Lizenzen zu einem Derivat führen kann, wohingegen ein Plugin, welches nur über einen *fork* Mechanismus aufgerufen wird, kein Derivat darstellt.
Es wird jede formale Lizenz um die Verbindungstypen erweitert und angegeben, ob es sich bei der Art der Verbindung um ein Derivat oder eine kombinierte Arbeit handelt. Handelt es sich um eine unabhängige Arbeit, kann der Verbindungstyp weggelassen werden. Dies soll die späteren Vergleiche zwischen den Lizenzen vereinfachen. Als Beispiel soll im Folgenden die Apache License Version 2 formal vorgestellt werden.

[7] www.gnu.org/licenses/gpl-3.0.rdf

```
<cc:License rdf:about="http://creativecommons.org/licenses/Apache-2.0/">
  <cc:permits rdf:resource="http://creativecommons.org/ns#Distribution"/>
  <cc:permits rdf:resource="http://creativecommons.org/ns#Reproduction"/>
  <cc:permits rdf:resource="http://creativecommons.org/ns#DerivativeWorks"/>
  <cc:requires rdf:resource="http://creativecommons.org/ns#Notice"/>
  <cc:requires rdf:resource="http://creativecommons.org/ns#Attribution"/>
  <cc:requires rdf:resource="http://creativecommons.org/ns#Patent"/>
  <cc:legalcode rdf:resource="http://opensource.org/licenses/Apache-2.0"/>
  <dc:title>Apache-2.0 License</dc:title>
  <dc:title xml:lang="de">Apache-2.0</dc:title>
  <dc:title xml:lang="en-us">Apache-2.0</dc:title>
  <dc:identifier>Apache</dc:identifier>
    <osr:staticLink>
      <rdf:Bag>
          <rdf:li rdf:resource="http://www.osrgroup.de/masterthesis#combined"/>
      </rdf:Bag>
  </osr:staticLink>
  <osr:dynamicLink>
      <rdf:Bag>
          <rdf:li rdf:resource="http://www.osrgroup.de/masterthesis#combined"/>
      </rdf:Bag>
  </osr:dynamicLink>
  <osr:fork>
      <rdf:Bag>
      </rdf:Bag>
  </osr:fork>
  <osr:ipc>
      <rdf:Bag>
      </rdf:Bag>
  </osr:ipc>
  <osr:linkedPlugin>
      <rdf:Bag>
          <rdf:li rdf:resource="http://www.osrgroup.de/masterthesis#combined"/>
      </rdf:Bag>
  </osr:linkedPlugin>
  <osr:forkPlugin>
  </osr:forkPlugin>
  <osr:subclass>
      <rdf:Bag>
          <rdf:li rdf:resource="http://www.osrgroup.de/masterthesis#derivative"/>
      </rdf:Bag>
  </osr:subclass>
</cc:License>
```

Abbildung 2.3: Formale Darstellung der Apache License Version 2 in ccREL

Besonders interessant in diesem Beispiel sind die Informationen zu den Verbindungstypen. Eine Komponente, welche statisch, dynamisch oder über ein Plugin verlinkt ist, stellt bei der Apache License Version 2 eine kombinierte Arbeit dar und somit greifen in diesem Fall auch die entsprechenden Anforderungen für kombinierte Arbeit aus *cc:require*. Wird für die Komponente die Verbindungsart *Subclass* verwendet, welche besagt, dass die Arbeit erweitert oder verändert wurde, so stellt dies ein Derivat dar und die entsprechenden Anforderungen für ein Derivat aus *cc:require* treten in Kraft.

Erweitertes Modell

Ein weiteres Modell, welches im Rahmen dieser Arbeit entwickelt wurde, basiert auf den Erkenntnissen von German (German & Hassan, 2009). Dort werden Lizenzen als Ansammlung von Rechten und Pflichten dargestellt. Dieser Aspekt wurde aufgegriffen und versucht die zehn beliebtesten Open Source Lizenzen[8]

[8]https://www.blackducksoftware.com/resources/data/top-20-open-source-licenses

und die Corel Transactional License (CTL)[9] als einzige proprietäre Lizenz, als Rechte und Pflichten zu modellieren.
Zunächst wurden alle Lizenztexte untersucht. Dabei wurde der gesamte Lizenztext analysiert, um festzustellen, welche Rechte und Pflichten existieren. Rechte stellen dabei immer Klauseln dar, die dem Nutzer der Lizenz ein bestimmtes Recht gewähren. Dies kann beispielsweise das Erstellen und Verbreiten eines Derivats sein. Pflichten sind dann die Klauseln, die vom Nutzer eingehalten werden müssen, damit er ein bestimmtes Recht erhält. Ein Beispiel hierfür ist, dass bei Erstellung und Verbreitung eines Derivats immer der Source Code mit verbreitet werden muss.
Die untersuchten Lizenzen wurden nun mittels eines eigens definierten XML Formates maschinenlesbar dargestellt. Dabei erhält jedes der extrahierten Rechte und Pflichten eine eindeutige ID. Die Rechte werden mit dem Element *Right* und die Pflichten mit dem Element *Obligation* modelliert. Bei der Untersuchung der Lizenzen fiel auf, dass es bestimmte Pflichten gibt, welche nicht alle eingehalten werden müssen, sondern stattdessen nur eine der aufgeführten Pflichten nötig ist. Diese Menge an Pflichten wurde durch ein *Selection* Element modelliert, sodass klar wird, dass es sich hierbei um eine Auswahl handelt. Als weiteres Merkmal fiel bei der Analyse auf, dass die vorhandenen Rechte und Pflichten teilweise nur für eine bestimmte Art der Komponentenverbindung zutreffen. Beispielsweise gibt es Rechte welche nur zutreffen, wenn es sich bei der lizenzierten Komponente um ein Derivat handelt oder um eine Kombination mehrerer Komponenten. Aus diesem Grund wurden die Lizenzen in vier Bereiche aufgeteilt, welche die möglichen Verbindungstypen darstellen. Diese sind *derivative*, *combined*, *usage* und *general*. Die Typen sind in der formalen Darstellung durch das *Operation* Element gekennzeichnet. Zur Unterscheidung enthält dieses Element jeweils noch das Attribut *type*, welches die beschriebenen Werte annehmen kann. Ein Grund für diesen Schritt war, dass es in den Lizenzen Pflichten gibt, die für mehrere unterschiedliche Rechte gelten. Da diese Rechte mit gleichen Pflichten wiederum verschiedenen Verbindungstypen zugeordnet sein können, musste hier eine Unterscheidung möglich sein. Ein weiterer Grund für die Aufteilung der Lizenzen in Bereiche war, dass nicht für jeden Verbindungstyp alle Klauseln untersucht werden müssen. Um die beschriebenen Aspekte zu veranschaulichen, wird in Abbildung 2.4 ein Ausschnitt aus der Artistic License gezeigt.

[9] http://apps.corel.com/clp/terms.html

```
<License name="Artistic">
    <Operation type="derivative">
        <Right id="1">may produce and distribute work based on the Program in source form</Right>
            <Obligation id="14">must include change notice</Obligation>
            <Selection id="21">
                <Obligation id="21a">must make modifications available to copyright holder</Obligation>
                <Obligation id="21b">must use in-house only</Obligation>
                <Obligation id="21c">use different name for non-standard executables.</Obligation>
            </Selection>
        <Right id="2">may produce and distribute work based on the Program in object form</Right>
            <Obligation id="14">must include change notice</Obligation>
            <Selection id="22">
                <Obligation id="22a">must distribute standard version and instructions where to get it.</Obligation>
                <Obligation id="22b">include source code with modifications</Obligation>
                <Obligation id="22c">must allow redistribution of modification under this or similar license</Obligation>
                <Obligation id="22d">must use different name for non-standard executables and include them and the standard executable</Obligation>
            </Selection>
    </Operation>
    <Operation type="combined">
        <Right id="3">may copy and distribute work in source form</Right>
            <Obligation id="11">must include copyright notice</Obligation>
            <Obligation id="12">must include disclaimer of warranty</Obligation>
        <Right id="4">may copy and distribute work in object form</Right>
            <Obligation id="27">include instructions how to get source</Obligation>
    </Operation>
```

Abbildung 2.4: Formale Darstellung der Artistic License in XML

2.3.4 Bestimmung der Operationen für Komponentenbeziehungen

Um einen Vergleich zwischen zwei Lizenzen durchführen zu können, muss immer auch die Art der Verbindung zwischen den lizenzierten Komponenten bestimmt werden, da diese entscheidet, welche Rechte und Pflichten der Lizenzen gültig sind. Es kann grundsätzlich zwischen drei verschiedenen Arten unterschieden werden, zusätzlich gibt es noch einen Bereich für allgemeine Rechte und Pflichten:

- **Derivat (derivative):** Derivat bedeutet, dass eine Modifikation des Originals stattgefunden hat. Analog zu dem Begriff *derivative work* aus dem Copyright.
- **Kombination (combination):** bedeutet, dass die Komponenten miteinander kombiniert werden und so zu einer Gesamtkomponente werden. Analog zu dem Begriff *combined work* aus dem Copyright.
- **Nutzung (usage):** Nutzung bedeutet, dass die Komponenten unabhängig voneinander genutzt werden und z.B. über eine Client-Server Verbindung miteinander kommunizieren.
- **Allgemein (general):** in dieser Kategorie werden alle Rechte und Pflichten gesammelt, die allgemeingültig sind und keiner bestimmten Operation zugeordnet werden können.

Wie bereits beschrieben, wurden die Klauseln der Lizenzen nach den Operationen eingeteilt, sodass eine bessere Unterscheidung möglich wird. Dabei wurde auch untersucht, welche Werte die einzelnen Klauseln annehmen können. Hierbei wurde entschieden, dass für die einzelnen Klauseln boolesche Werte genügen. Somit genügt es, wenn bei der Modellierung der Lizenzen die Klauseln angegeben

werden. Diese werden dann als wahr betrachtet und alle nicht Vorhandenen als falsch.

2.3.5 Regeln für die Operationen

Nachdem im vorherigen Abschnitt die möglichen Operationen zwischen Komponenten mit unterschiedlichen Lizenzen erläutert wurden, muss nun noch definiert werden, wie ein Vergleich zwischen Lizenzen auf Basis dieser Operationen erfolgen soll. Es wurden zunächst alle möglichen Klauseln aus den untersuchten Lizenzen gesammelt und anschließend die Gleichen aussortiert. Die übrigen individuellen Klauseln wurden nun auf die vier verschiedenen Operationen aufgeteilt. Eine Übersicht aller Klauseln ist in Anhang C zu finden.
Anschließend wurden für die einzelnen Klauseln Regeln definiert, welche festlegen wie sich diese Klausel in Zusammenhang mit anderen Lizenzen und bezogen auf die Operation verhält. Eine Annahme, die dabei getroffen wurde, ist, dass es für die Verbindung zwischen den Lizenzen immer eine Richtung gibt. Das bedeutet, wenn man Komponente A und Komponente B mit Lizenz A und Lizenz B hat und diese mittels einer der Operationen, z.B. Derivat, miteinander verbunden sind, dann beinhaltet A ein Derivat von B und ist als Gesamtes mit Lizenz A lizenziert. Somit wird immer für Lizenz B überprüft, ob diese ein Derivat erlaubt und ob dieses in A mit Lizenz A integriert werden kann. Analog erfolgt die Überprüfung für die anderen Operationen.
Die Modellierung der Regeln erfolgt genauso wie die Lizenzen mittels XML. Da die Klauseln nur boolesche Werte annehmen können, erfolgt die Modellierung der Regeln als logische Ausdrücke. Diese werden verschachtelt in XML dargestellt. Startelement ist dabei immer *Rule*. Darauf folgt dann ein Ausdruck, der entweder atomar oder binär sein kann. Modelliert wird dies über das Element *Expr*, welches ein Attribut *type* besitzt. Dieses Attribut gibt an, um welche Art es sich bei dem Ausdruck handelt. Mögliche Werte sind *And*, *Or*, *Atomic*, *Equals* und als Spezialfall *All*. Sollte es sich um einen atomaren Ausdruck handeln, so hat dieser noch die Angabe, für welche Lizenz (*License*) er gilt (A oder B). Außerdem gibt es dann die Möglichkeit, eine Referenz (*Ref*) anzugeben, wenn der Ausdruck nicht für die Klausel gilt, die der Regel zugeordnet ist. Für binäre Ausdrücke gibt es eine rechte (*Right*) und eine linke (*Left*) Seite, welche wiederum einen Ausdruck beinhalten. Die Darstellung der Regeln für die Operation Derivat ist in Abbildung 2.5 anhand eines Beispiels zu sehen.

```
<Rule id="2">
    <Expr type="Or">
        <Left>
            <Expr type="Atomic">
                <License>B</License>
            </Expr>
        </Left>
        <Right>
            <Expr type="Atomic">
                <License>B</License>
                <Ref>1</Ref>
            </Expr>
        </Right>
    </Expr>
</Rule>
<Rule id="11">
    <Expr type="Atomic">
        <License>B</License>
    </Expr>
</Rule>
```

Abbildung 2.5: Formale Darstellung zweier Regeln für ein Derivat

2.3.6 Funktionsweise der Lizenzalgorithmen

Nachdem nun die Architektur und die Lizenzen modelliert sind, wird noch eine Möglichkeit benötigt, die Lizenzen untereinander zu vergleichen. Hierfür wurde für jedes Modell ein Lizenzalgorithmus entworfen. Die theoretischen Überlegungen sind im Folgenden dokumentiert. Die Beschreibung der Implementierungen dieser Algorithmen findet sich im Abschnitt 3.3.

Modell basierend auf ccREL
Für den Vergleich von Lizenzen ist einiges an Denkarbeit notwendig. Die Unterscheidung zwischen Derivaten und kollektiver Arbeit spielt eine große Rolle. Hierzu gehört auch, wie die Komponenten wiederverwendet werden. Es ist wichtig zu unterscheiden, wie zwei Komponenten mit unterschiedlicher Lizenz miteinander in Verbindung stehen und wie sich diese Verbindung auf die Lizenzierung auswirkt. Ein weiterer Punkt ist, wie sich die "Richtung" auf den Lizenzvergleich auswirkt. Es gibt einige Lizenzen, bei denen es möglich ist, Komponenten, welche unter *Lizenz A* stehen, in andere Komponenten zu integrieren, welche unter der *Lizenz B* stehen, jedoch nicht umgekehrt.
Es können immer zwei Lizenzen miteinander verglichen werden. Um ganze Hierarchien auf Unstimmigkeiten zu überprüfen, muss von unten nach oben vorgegangen werden. Es wird immer überprüft, ob die Blätter eines Knotens mit der Lizenz des Knotens übereinstimmen. Das wird so lange fortgeführt, bis man an der Wurzel angekommen ist.
Rechte (Permission) werden 1 zu 1 überprüft. Damit Lizenzen kompatibel sind, müssen sie dieselben Rechte haben.

Für die Bedingungen (Requirement) wurden Regeln definiert, um festzulegen, welche Kombinationen erlaubt sind und welche nicht.
Besonders wichtig ist dabei zu entscheiden, wie sich die Regeln verhalten, wenn sie für die eine Seite definiert sind und für die andere nicht.
Hat eine Lizenz die *Share Alike* Voraussetzung und die andere nicht, passen diese, sofern es sich um ein Derivat handelt, nicht zusammen.
Hat eine Lizenz die *Copyleft* Voraussetzung und die andere nicht passen diese, sofern es sich um ein Derivat oder eine kombinierte Arbeit handelt, nicht zusammen.
Hat eine Lizenz die *Lesser Copyleft* Voraussetzung und die andere nicht, passen diese, sofern es sich um ein Derivat handelt nicht zusammen. Es gilt die Ausnahme wenn die andere Lizenz die *CopyLeft* Voraussetzung hat.
Hat eine Lizenz die *Source Code* Voraussetzung und die andere nicht, passen diese nicht zusammen, sofern es sich um ein Derivat oder eine kombinierte Arbeit handelt.
Hat eine Lizenz die *Network Source* Voraussetzung und die andere nicht, passen diese nicht zusammen, sofern es sich um ein Derivat oder eine kombinierte Arbeit handelt.
Hat eine Lizenz die *Patent* Voraussetzung und die andere nicht, passen diese nicht zusammen, sofern es sich um ein Derivat oder eine kombinierte Arbeit handelt.
Wichtig für diese Definitionen ist auch die "Richtung" in welche sie interpretiert werden. Es muss beachtet werden, dass diese Regeln nur dann gelten, wenn eine Komponente mit einer strengen Lizenz (z.B. GPLv3) in einer Komponente mit einer toleranten Lizenz (z.B. Apache License Version 2) eingebaut wird. Gemeint ist damit, dass z.B. die CopyLeft Bedingung nur zu Problemen führt, wenn eine Komponente mit GPLv3 in einer Komponente mit z.B. Apache License Version 2 verwendet werden soll. In der anderen Richtung wäre dies kein Problem.
Im Folgenden finden sich zwei Tabellen, welche die Regeln für die jeweilige Richtung zeigen. Diese Regeln sind ähnlich der Regeln aus (Villata & Gandon, 2012), obwohl beide Arbeiten unabhängig voneinander entwickelt wurden.
In der ersten Tabelle enthält die Komponente mit der Lizenz A die Komponente mit der Lizenz B entweder als Derivat oder als kombinierte Arbeit:

Regel	Kompatibilität
A hat Attribution, B hat Attribution nicht	Kompatibel
A hat Notice, B hat Notice nicht	Kompatibel
A hat Share Alike, B hat Share Alike nicht	Kompatibel.
A hat Source Code, B hat Source Code nicht	Inkompatibel
A hat CopyLeft, B hat CopyLeft nicht	Kompatibel
A hat Lesser CopyLeft, B hat LesserCopyLeft nicht	Kompatibel
A hat Patent, B hat Patent nicht	Kompatibel
A hat NetworkSource, B hat NetworkSource nicht	Kompatibel

Tabelle 2.3: Kompatibilität von A → B

In der zweiten Tabelle ist dargestellt, wie es sich für die andere Richtung verhält, nämlich aus der Sicht der Komponente mit Lizenz B, welche ein Teil der Komponente mit Lizenz A ist.

Regel	Kompatibilität
B hat Attribution, A hat Attribution nicht	Kompatibel
B hat Notice, A hat Notice nicht	Kompatibel
B hat Share Alike, A hat Share Alike nicht	Inkompatibel (bei derivative work)
B hat Source Code, A hat Source Code nicht	Inkompatibel
B hat CopyLeft, A hat CopyLeft nicht	Inkompatibel (bei derivative und collective work)
B hat Lesser CopyLeft, A hat Lesser CopyLeft nicht	Inkompatibel wenn Derivat. Ausnahme, wenn A CopyLeft hat.
B hat Patent, A hat Patent nicht	Inkompatibel
B hat NetworkSource, A hat NetworkSource nicht	Inkompatibel

Tabelle 2.4: Kompatibilität von B → A

Aus diesen Tabellen kann man sehr gut ablesen, welchen Unterschied die Richtung der Verbindung ausmacht.

Erweitertes Modell

In den bisherigen Abschnitten zum erweiterten Modell wurde erläutert, wie eine Lizenz und die Operationen für den Vergleich von Lizenzen modelliert wurden. Nun muss noch definiert werden, wie der Vergleich für das erweiterte Modell im Detail ablaufen soll und welche Einschränkungen angenommen wurden.
Die praktische Umsetzung der Überlegungen aus diesem Abschnitt werden dann in 3.3 besprochen.
Ziel war es, für den Vergleich der Lizenzen eine allgemeine Lösung zu finden, die auf alle Operationen angewandt werden kann.
Wie bereits beschrieben, haben die Beziehungen zwischen den Komponenten immer eine Richtung. Diese Richtung spielt für den Vergleich eine entscheidende Rolle, da sie die Hierarchie der Komponenten darstellt und deshalb mit Hilfe der Richtung bestimmt werden kann, auf welche Komponente die Operationen angewandt werden müssen. Beispielsweise bedeutet es bei einer Derivat-Beziehung zwischen zwei Komponenten (A nach B), dass A das Derivat von B beinhaltet und somit das Derivat von B durch die Integration in A auch dessen Lizenz erhält. Für diesen Fall wird überprüft, ob für die Lizenz von B ein Derivat möglich ist und ob es möglich ist dieses unter A zu veröffentlichen.
Der Ablauf hierfür ist, dass für beide Komponenten die Lizenzen und der Verbindungstyp bestimmt werden muss. Anschließend werden die beiden Lizenzen und die Regeln für die Operation, welche diesem Verbindungstyp zugeordnet ist, geladen. Danach werden die Rechte und Pflichten von Lizenz B für den Verbindungstyp durchlaufen und für jede dieser Klauseln die Regeln bestimmt. Diese werden nun auf ihre Gültigkeit überprüft, indem der logische Ausdruck der Regel ausgewertet wird. Nachdem alle Regeln überprüft wurden, wird zum Schluss noch das Gesamtergebnis bestimmt. Nähere Details zum Ablauf sind im Abschnitt 3.3 beschrieben.

3 Implementierung

In diesem Kapitel wird beschrieben, wie die Konzepte und Modelle aus dem vorherigen Kapitel praktisch umgesetzt wurden. Zunächst wird die Architektur des Editors erklärt und erläutert, wie die einzelnen Bestandteile aufgebaut sind. Anschließend erfolgt die Beschreibung, wie die Anwendung mit dem externen FOSSology Server verknüpft wurde und worauf dabei zu achten war. Abschließend wird die Erklärung zu der Implementierung der beiden Lizenzmodelle und deren Lizenzalgorithmen dargestellt.

3.1 Entwurf und Architektur des Editors

Einer der Hauptbestandteile dieser Arbeit ist der Entwurf und die Implementierung eines Editors für Softwarearchitekturen. Wie bereits im Abschnitt 2.3.1 erläutert, soll dieser Editor eine Softwarearchitektur darstellen können und es ermöglichen, diese auch zu bearbeiten. Für den Editor wurde auf das *Google Web Toolkit (GWT)* zurückgegriffen. (*Google Web Toolkit*, 2015)

Google Web Toolkit (GWT)
GWT ist ein Entwicklungswerkzeug von Google, mit dem komplexe Browseranwendungen erstellt werden können. Das Ziel von GWT ist es, komplexe Webentwicklung auch für Entwickler zur Verfügung zu stellen, die keine Experten in JavaScript und weiteren Webentwicklungswerkzeuge sind.
Bei GWT kann die vollständige Logik in Java geschrieben werden. Dieser Java-Code wird dann von GWT in JavaScript übersetzt und optimiert. Es ist aber trotzdem möglich weiterhin in JavaScript zu programmieren. Hierfür kann mittels speziellen Kommentaren der JavaScript-Code direkt in den Java-Code eingebaut werden.
Die Gestaltung der Weboberflächen kann bei GWT entweder direkt im Code vorgenommen werden oder man verwendet den eigens dafür entwickelten *UiBinder* von GWT. (*UiBinder*, 2009) Der UiBinder erlaubt es für die Entwicklung des *User Interfaces* (UI) auf die Standardwerkzeuge *HTML* und *CSS* zurückzugreifen,

anstatt dieses im Java-Code entwickeln zu müssen. Durch den Einsatz von *UiBinder* kann man im HTML Code die GWT eigenen Elemente, *GWT-Widgets* genannt, einbinden. Das bringt die Vorteile, das die UI viel besser abgrenzbar von der Logik ist. Des Weiteren fördert es die Zusammenarbeit mit Designern, welche es gewohnt sind, die Webseiten in HTML und CSS zu entwerfen. Durch *UiBinder* ist es außerdem einfacher, bereits bestehende Webseiten auf GWT umzustellen, da der HTML Code weitestgehend übernommen werden kann und nur durch die GWT-Widgets ergänzt werden muss. Dies ist viel einfacher als die Webseite von Neuem in Java-Code zu schreiben.

Architektur des Editors

Das in dieser Arbeit entwickelte Werkzeug ist aufgeteilt in einen Client und einen Server. Der Client beinhaltet die Webanwendung, mit der der Nutzer interagiert. Hier kann sich der Nutzer eine Softwarearchitektur anzeigen lassen oder eine bestehende Architektur editieren und erweitern.
Der Aufbau folgt dabei dem *Model-View-Presenter* (MVP) Muster, welches Google für GWT Anwendungen empfiehlt.(Ramsdale, 2010)
Ein Grund dafür ist, dass das MVP Muster die Entwicklung auf eine Art entkoppelt, dass mehrere Entwickler gleichzeitig problemlos an einer Anwendung entwickeln können.
Der Grundgedanke des MVP Musters ist die Aufteilung der Funktionalität auf Komponenten, die logisch Sinn machen. Bei GWT ist das Hauptziel, die Ansichten (Views) so simpel wie möglich zu gestalten. Dies ist im Besonderen auch wichtig, wenn man Tests schreiben möchte, da Ansichten schwieriger zu testen sind. Ein Grund dafür ist, dass sie immer auch JavaScript Code enthalten und um diesen zu testen, werden spezielle Tests benötigt. Wird die Logik jedoch ausgegliedert, kann dieser Teil mit ganz normalen *JUnit* Tests getestet werden.(Ramsdale, 2010)
Bei dem MVP Muster hat man einen *Presenter* welcher zwischen dem *View* und dem *Model* vermittelt. Er sorgt z.B. dafür, dass bei Änderungen im View das Modell aktualisiert wird und umgekehrt. Dies ist in folgender Abbildung noch einmal schematisch dargestellt:

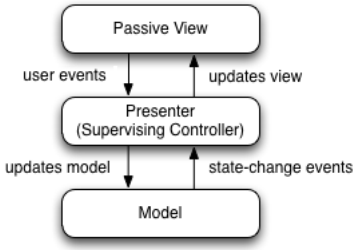


Abbildung 3.1: Darstellung des MVP-Musters (*MVP*, 2015)

Um in der Webanwendung einen Verlauf, wie er für Webseiten üblich ist, zu realisieren, wird auf das *Activities and Places Framework* von *GWT* zurückgegriffen. Das bereits vorgestellte MVP Muster und das Activities and Places Framework wurden in der hier vorliegenden Arbeit zusammen verwendet und ergeben die im folgenden vorgestellte Architektur:
Es werden Schnittstellen für die einzelnen Ansichten (View) angelegt und dafür konkrete Implementierungen bereitgestellt, welche sich um die Darstellung der Anwendung kümmern. Die Schnittstellen enthalten außerdem eine weitere *Presenter* Schnittstelle, welche eine bidirektionale Kommunikation zwischen der Ansicht und der Aktivität (Activity), welche den Presenter implementiert, erlaubt. Dies ist sinnvoll, weil dadurch UiBinder verwendet werden kann. Auf diese Art können die von UiBinder bereitgestellten UiHandler Methoden, welche für den Zugriff auf GWT-Widgets benötigt werden und nur in den Ansichten eingesetzt werden können, die Daten an den Presenter weiterleiten.
Für jede Ansicht wird eine Aktivität definiert, welche eine Referenz auf die Ansicht erhält und die Presenter Schnittstelle implementiert. Die Aktivitäten erhalten außerdem zusätzlich einen zugehörigen Ort, in GWT *Place* genannt, welcher für den Browserverlauf notwendig ist. Die Orte stellen die verschiedenen Zustände der Webanwendung dar, zwischen denen hin und her gewechselt werden kann.
Des Weiteren wird ein *PlaceHistoryMapper* benötigt, der alle Orte definiert, und ein *ActivityMapper*, welcher die Aktivitäten und die Orte zueinander zuweist.
Das Zusammenspiel sieht dann folgendermaßen aus: Der *ActivityMapper* überwacht alle Aktivitäten, die im aktuellen Kontext vorhanden sind. Wird in der Webanwendung nun zu einem neuen Ort navigiert, so wird die aktuelle Aktivität darüber informiert und sollte diese einen Ortswechsel erlauben, wird über den *ActivityMapper* die Aktivität für den neuen Ort gesucht und geladen. Dies stellt den vereinfacht beschriebenen Ablauf dar.
Das MVP Muster ist dabei dafür verantwortlich, dass die ganze Logik in den Aktivitäten vorhanden ist und die Ansichten sich nur um die Darstellung kümmern. Das bedeutet, dass die Aktivitäten die *Presenter* aus dem MVP Muster darstellen.
Die zu dieser Arbeit entwickelten Ansichten und Aktivitäten und ihr Zusammenspiel sind in Anhang D zur Veranschaulichung als Klassendiagramm zu finden.

Aufbau der UI

Der Editor hat mehrere verschiedene Aufgaben. Zunächst einmal muss eine Darstellung der Softwarearchitektur als zyklischer Abhängigkeitsgraph erfolgen. Für die Visualisierung dieses Graphen wurde in dieser Arbeit *gwt-d3*[1] verwendet. *gwt-d3* ist eine Bibliothek, die einen Zugriff auf die *D3.js*[2] API für GWT zur Verfügung stellt. Sie ermöglicht es, die JavaScript Bibliothek *D3.js* in GWT zu

[1] https://github.com/gwtd3/gwt-d3
[2] http://d3js.org/

verwenden, ohne dass JavaScript nötig ist. Ein Beispiel für eine Visualisierung und das grundlegende Design des Editors ist in Abbildung 3.2 zu sehen.

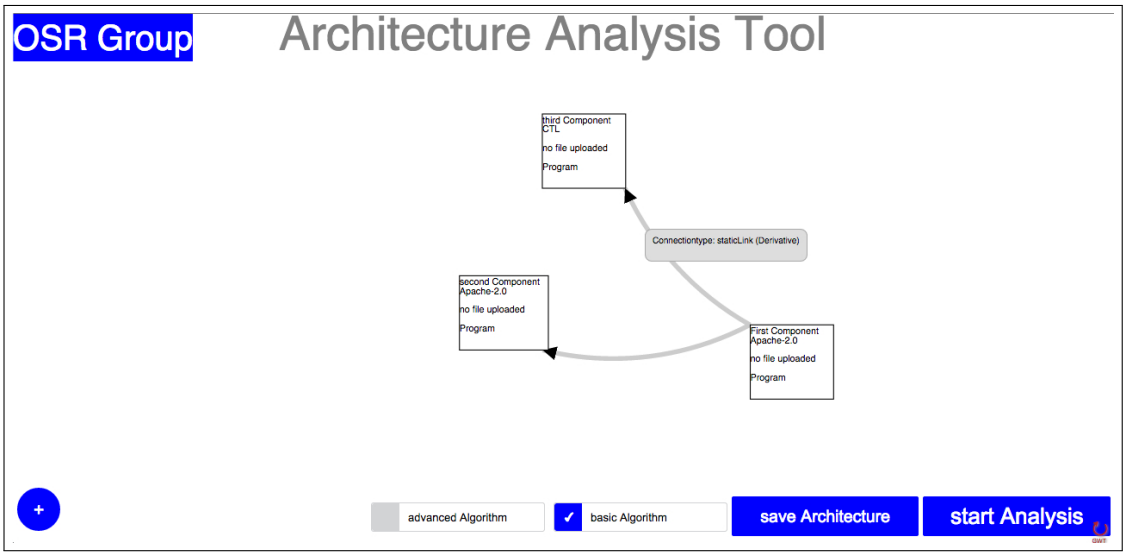


Abbildung 3.2: Architekturansicht der Architecture Analysis Application

Dabei ist zu beachten, dass nähere Informationen zu den Verbindungstypen und mögliche Konflikte angezeigt werden, sobald der Nutzer mit der Maus über eine Verbindung geht. Dieses Fenster mit den zusätzlichen Informationen, ist in der Abbildung neben dem Graph zu erkennen. Des Weiteren ist zu sehen, dass der Nutzer die Möglichkeit hat, über den Button *save Architecture* die aktuelle Architektur, in dem aus dieser Arbeit bekannten ADL Format abzuspeichern.
Neben der Visualisierung einer Softwarearchitektur muss es auch möglich sein, diese zu erweitern. Hierfür ist in Abbildung 3.2 links unten das blaue Plus. Wird dieses ausgewählt, wechselt der Editor in eine neue Ansicht, zu sehen in Abbildung 3.3, in der man eine neue Komponente oder eine neue Verbindung hinzufügen kann. Hierfür kann man verschiedene Optionen auswählen. Die möglichen Werte für eine neue Komponente sind in Tabelle 3.1 aufgelistet, die für eine neue Verbindung in Tabelle 3.2. Sie sind außerdem in Abbildung 3.3 auf der linken Seite zu erkennen. Im oberen Bereich sind die Optionen für eine neue Komponente zu sehen, im unteren jene für eine neue Verbindung. Mit den beiden Buttons *create new Node* und *create new Link* können neue Komponenten oder neue Verbindungen erstellt werden.

Option	Werte
Name	Name der neuen Komponente
Beschreibung	Beschreibung für die neue Komponente
Verbindungstyp	static link, dynamic link, fork, ipc, linked plugin, fork plugin, subclass
Lizenz	hier kann eine der 10 Open Source Lizenzen oder die CTL gewählt werden
Lizenztyp	proprietär, foss
Quelle	Hier kann die Quelle für die Verbindung gewählt werden.
Komponententyp	Program, Library
Operation	Derivative, Integration, Usage

Tabelle 3.1: Mögliche Werte für neue Komponente

Option	Werte
Beschreibung	Beschreibung für die Verbindung
Quelle	Quelle für die Verbindung
Ziel	Ziel für die Verbindung
Verbindungstyp	static link, dynamic link, fork, ipc, linked plugin, fork plugin, subclass
Operation	Derivative, Integration, Usage

Tabelle 3.2: Mögliche Werte für neue Verbindung

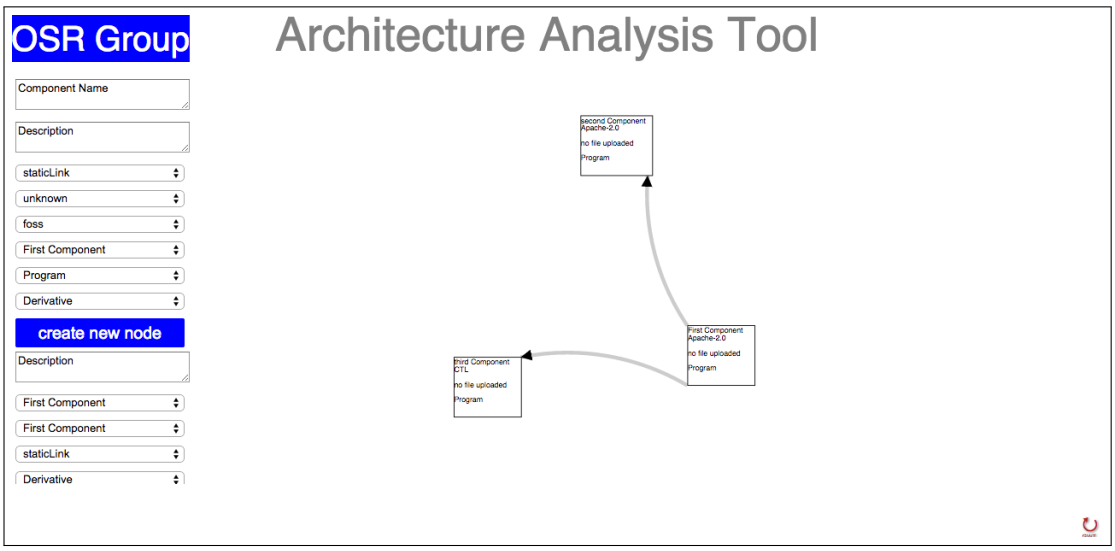


Abbildung 3.3: Ansicht "neue Komponente oder Verbindung hinzufügen"

Weitere Möglichkeiten sind die Veränderung einer bereits bestehenden Architektur. Um eine Komponente editieren zu können, muss diese nur angeklickt werden und der Editor wechselt in die Ansicht in Abbildung 3.4. In dieser Ansicht werden auf der linken Seite die aktuellen Eigenschaften der ausgewählten Komponente angezeigt. Diese Eigenschaften können beliebig verändert werden und werden dann über den Button *save* gespeichert. Außerdem kann der Nutzer für die gewählte Komponente eine Datei mit dem Source Code hochladen, um diese vom FOSSology Server analysieren zu lassen. Hierfür ist der Button *choose a file to upload* vorgesehen. Über den Button *delete* kann die ganze Komponente entfernt werden. Dies ist jedoch nur möglich wenn die Komponente keine ausgehende Verbindung hat, da sonst ganze Zweige unwiderruflich entfernt werden.
Möchte man eine Verbindung ändern, kann man dies tun, indem man auf die Verbindung klickt. Anschließend wechselt der Editor in die Ansicht in Abbildung 3.5. In dieser Ansicht kann wiederum auf der linken Seite die Verbindung editiert werden und über den Button *save* gespeichert oder über den Button *delete* gelöscht werden.

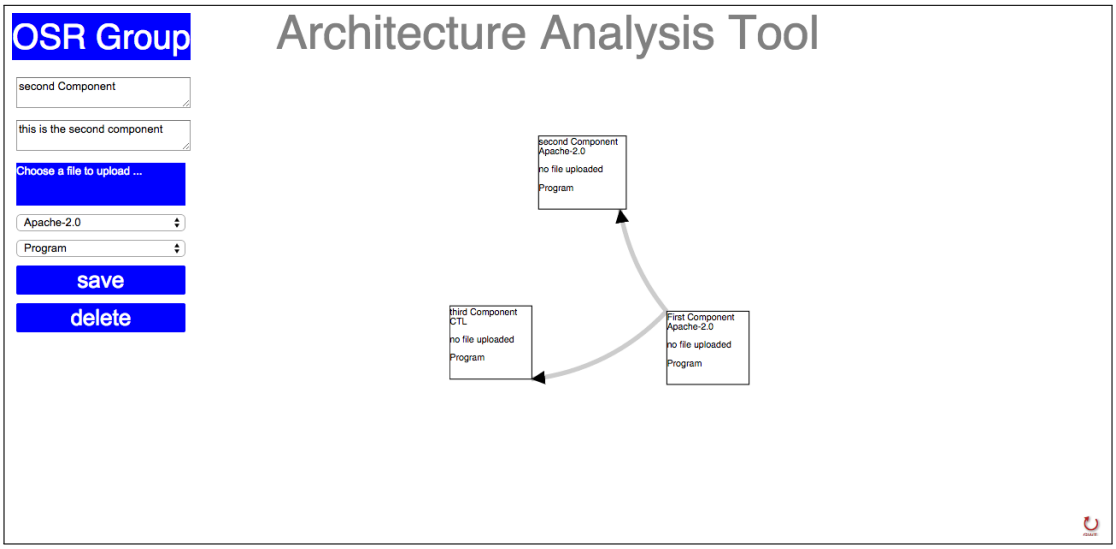


Abbildung 3.4: Ansicht "Komponente editieren und Datei hochladen"

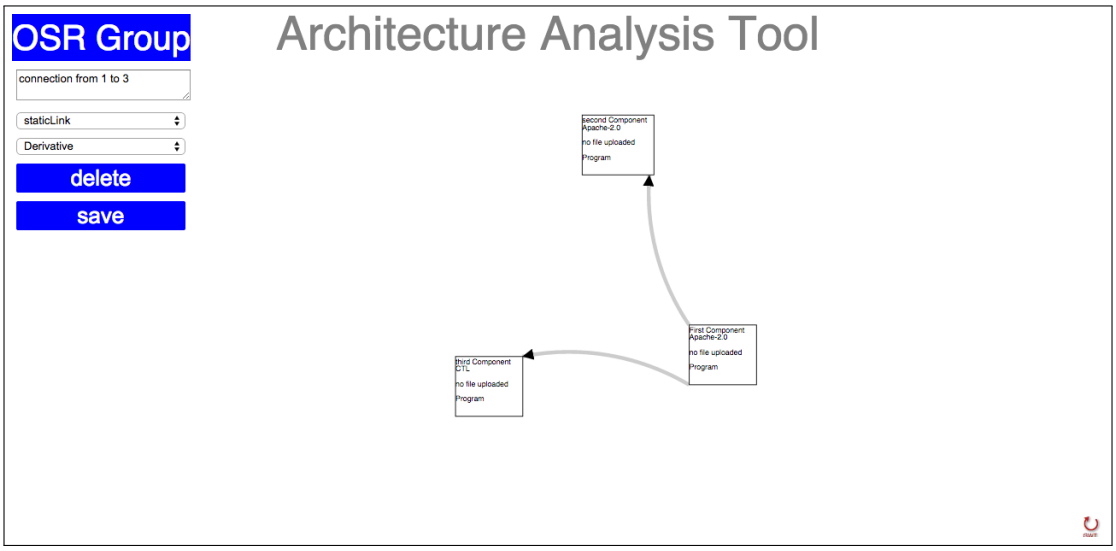


Abbildung 3.5: Ansicht "Verbindung editieren"

Anbindung an den Server

Um die im Editor erstellte Architektur analysieren zu können, muss der Client mit dem Server verbunden werden. Diese Verbindung erfolgt über *Java-Servlets*. Der Austausch von Daten zwischen dem Client und dem Server erfolgt über ein eigens dafür definiertes *JavaScript Object Notation* (JSON)-Format. Dieses JSON-Format ist unterteilt in Komponenten und Verbindungen, sodass jede Art von Graph damit realisiert werden kann. Der Aufbau des JSON-Formats ist hier in einem Auszug aus einem Beispiel dargestellt:


```
{
"nodes": [
  {
  "ComponentId": "Component1",
  "name": "First Component",
  "description": "this is the first component",
  "license": "Apache-2.0",
  "licenseType": "foss",
  "fileName": "null"
  "componentType": "Program"
  }]
```

```
"links": [
  {
  "id": "Connection1",
  "description": "connection one",
  "source": "Component1",
  "target": "Component2",
  "conflict": "no",
  "conflictDescription": "there is no conflict",
  "connectionType": "staticLink"
  "relationshipType": "Derivative"
  }]
},
```

Für die Zuordnung von Verbindungen zu den jeweiligen Komponenten werden in den Verbindungen die IDs für die Ausgangs- und Zielkomponente gespeichert. Ansonsten enthalten die Komponenten die jeweiligen Informationen, die vorher für die Architektur definiert wurden. Die Verbindungen enthalten zusätzlich Informationen zu den Konflikten (*conflictDescription*) und die beiden Verbindungstypen. Diese unterscheiden sich darin, dass *connectionType* die Verbindungstypen aus Softwarearchitektur Sicht beinhalten und *relationshipType* die Verbindungsarten aus lizenzrechtlicher Sicht.
In Anhang E befindet sich das vollständige Beispiel einer JSON-Datei, wie sie in dieser Anwendung vorkommen kann.

3.2 REST-Anbindung von FOSSology

Um eine möglichst lose Kopplung zwischen der in dieser Arbeit entwickelten Anwendung und dem FOSSology Dienst zu schaffen, wurde dieser über eine dafür entwickelte REST-Schnittstelle an den Server angebunden. Dargestellt ist dies in der Abbildung 3.6. Diese Art der Anbindung hat den Vorteil, dass ein Austausch von FOSSology mit einem anderen Analysewerkzeug einfacher zu realisieren ist. Hierfür müsste nur eine neue Schnittstelle entwickelt werden oder die bereits bestehende an das neue System angepasst beziehungsweise erweitert werden. Ein weiterer Vorteil entsteht im lizenzrechtlichen Bereich. FOSSology wurde von Hewlett-Packard unter der *GPLv2* Lizenz veröffentlicht. Diese Lizenz zählt zu den sogenannten propagierenden Lizenzen und hat erheblichen Einfluss auf die Weiterverbreitung der Software, in welcher sie verwendet wird. Es muss z.B. jegliche Software, die eine Komponente unter der Lizenz verwendet, bei einer Veröffentlichung dieselbe Lizenz haben wie die Komponente. Dies nennt man die *Copyleft* Klausel, welche unter der GPLv2 dazu führt, dass jegliche Arbeit, welche

die unter GPLv2 stehende Arbeit enthält, egal ob in modifizierter oder Originalversion, genauso wieder unter der GPLv2 veröffentlicht werden muss. Wird die Komponente jedoch nur über einen Server angebunden, sodass ein Client auf diese zugreifen kann, so greift diese Lizenzregelung nicht, da die Komponente dadurch nicht weiterverbreitet wird. Daraus folgt, dass durch die externe Verwendung von FOSSology kein Einfluss auf die Lizenzregelung der hier entwickelten Lösung entsteht.

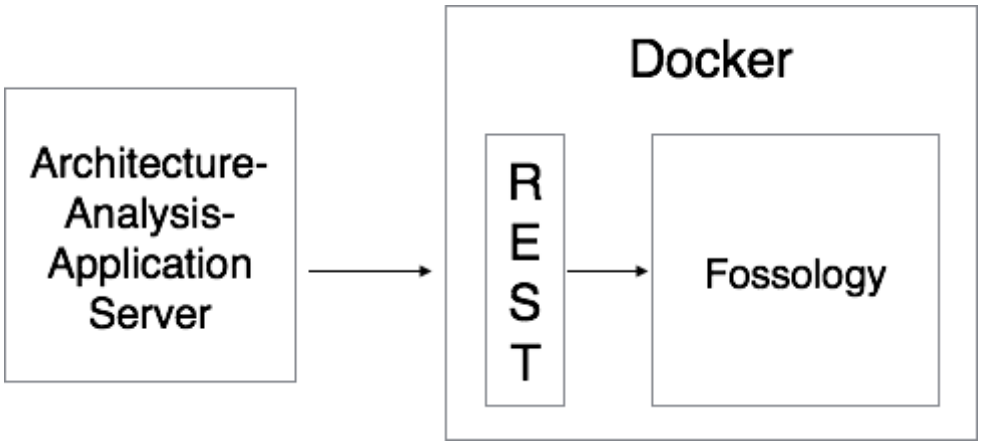


Abbildung 3.6: Anbindung von FOSSology an Server-Komponente

FOSSology wird in der Regel als Webanwendung verwendet und hat somit einen Großteil seiner Funktionalität über diese Schnittstelle. Es liegt jedoch auch eine Schnittstelle über die Kommandozeile vor. Über diese Schnittstelle ist es möglich, Dateien an den FOSSology Server zu senden und die Ergebnisse entgegen zu nehmen. Hierfür gibt es folgende Kommandos:

- **cp2foss:** mit *cp2foss* kann man eine oder mehrere Dateien an den FOSSology Server hochladen. Mit der Option *-f* kann dabei der Ordner festgelegt werden, in welchem die Dateien gespeichert werden. Hochgeladene Archive können eine Datei oder eine URL zu der Datei sein. Sollte ein Archiv ein Ordner sein, werden alle Dateien unter diesem Ordner hinzugefügt. (*cp2foss*, 2015)
- **fossjobs:** kann verwendet werden, um die Agenten von FOSSology aufzulisten. Des Weiteren können Analysen von Dateien, mit bestimmten Agenten, von *fossjobs* geplant werden. Dafür muss bei fossjobs die *upload id* der zu analysierenden Datei mitgegeben werden. (*fossjobs*, 2015)

- **fo_nomos_license_list:** der fo_nomos_license_list agent lässt den nomos Agenten in Echtzeit laufen. Dafür verwendet er Dateien, die bereits vorher in FOSSology hochgeladen wurden. (*fo_nomos_license_list*, 2015)

Die hier entwickelte Softwarelösung verwendet diese Kommandos, um für die vom Nutzer hochgeladenen Dateien eine Analyse durchzuführen. Zunächst wird das Kommando cp2foss verwendet, um die hochgeladenen Dateien an FOSSology weiterzureichen. Der Befehl ist:

cp2foss --username fossy --password fossy -f folder -q agent_nomos fileName

Anschließend kann die Analyse mit dem Befehl fossjobs ausgeführt werden.

fossjobs --user fossy --password fossy -A agent_nomos -U uploadPk

Um die Ergebnisse über die Kommandozeile zu erhalten, benötigt man zusätzlich den dritten Befehl fo_nomos_license_list

fo_nomos_license_list --user fossy --password fossy -u uploadPk

Die Ergebnisse, welche FOSSology zurückliefert, sind in einem *Comma-Separated-Value* (CSV) Format und haben folgenden Aufbau:

Paket/Dateiname, Lizenz

Es werden jedoch keine weiteren Informationen bezüglich der Zusammensetzung des hochgeladenen Archivs übermittelt. Für die weitere Analyse der Softwarearchitektur wäre es hilfreich, wenn eine genauere Unterteilung nach Subkomponenten möglich wäre oder ein Hinweis auf referenzierte Komponenten. Auf Grund dieser fehlenden Informationen, wird in dieser Arbeit aus den von FOSSology zurückgegebenen Lizenzen, für eine Komponente, jene ausgesucht, welche am häufigsten vorkommt. Dies wird dann als die Lizenz der Komponente angenommen.

3.3 Implementierung der Lizenzalgorithmen

Modell basierend auf ccREL
Der Lizenzalgorithmus ist dafür da, Lizenzkonflikte zu erkennen. Er verwendet die in Abschnitt 2.2.2 vorgestellte Architektur und das Modell einer Lizenz, um auf Basis dieser Eingaben zu bestimmen, ob ein Konflikt vorliegt oder nicht. Für die Bestimmung eines Lizenzkonfliktes werden die in Abschnitt 2.3.6 vorgestellten Regeln verwendet.
Zunächst wird hier erläutert, wie das Modell einer Lizenz praktisch umgesetzt wurde. Anschließend wird gezeigt wie auf Basis dieser Umsetzung der Lizenzalgorithmus arbeitet.

Praktische Umsetzung des Lizenzmodells
Unter Abschnitt 2.3.3 wurde bereits die maschinenlesbare Darstellung einer Lizenz beschrieben. Diese Darstellung wird mittels eines Parsers, basierend auf *Apache Jena*[3], einem Framework zum Einlesen und Schreiben von RDF Dateien, eingelesen. Die Zuordnung von Modell zu Java-Klassen sieht dabei folgendermaßen aus:
Es gibt eine allgemeine Lizenzklasse, welche folgende Informationen enthält:

- **Permissions**: eine Liste aus String Objekten mit den Genehmigungen.
- **Prohibitions**: eine Liste aus String Objekten mit den Verboten.
- **Requirements:** eine Liste aus Bedingungsobjekten. Es gibt dabei eine Schnittstelle, welche von allen konkreten Bedingungen implementiert wird. Diese wurden bereits im Abschnitt 2.3.3 beschrieben.
- **AnalysisConnectionTypes**: eine Liste aus *AnalysisConnectionType* Objekten. Diese Objekte enthalten die jeweiligen Objekte für die Verbindungstypen und die Information, ob es sich in diesem Fall um ein Derivat oder eine kombinierte Arbeit handeln kann. Die möglichen Verbindungstypen wurden bereits im Abschnitt 2.3.3 beschrieben.
- **Lizenzname**
- **Lizenzidentifikator**: eindeutiger Identifikator für die Lizenz.
- **Version**

Der Algorithmus zur Überprüfung der Kompatibilität zwischen zwei Lizenzen ist im Folgenden als Pseudocode dargestellt:

[3]https://jena.apache.org/

$conflict \leftarrow false$
if $LizenzA == LizenzB$ **then**
  if $VersionA \neq VersionB$ **then**
    if *connectionType.isDerivative* **or** *connectionType.isCombined* **then**
      $conflict \leftarrow true$
    end if
  else
    $conflict \leftarrow false$
  end if
end if

if *permA.containsAll*(*permB*) **and** *permB.containsAll*(*permA*) **then**
  $conflict \leftarrow false$
else
  $conflict \leftarrow true$
end if

if *prohibA.containsAll*(*prohibB*) **and** *prohibB.containsAll*(*prohibA*) **then**
  $conflict \leftarrow false$
else
  $conflict \leftarrow true$
end if

if $hasCopyLeft(requirementsA)$ **and** $hasCopyLeft(requirementsB)$ **then**
  if $LizennameA \neq LizennameB$ **then**
    if *connectionType.isDerivative* **or** *connectionType.isCombined* **then**
      $conflict \leftarrow true$
    end if
  end if
end if

if $hasLesserCopyLeft(requirementsA)$ **and** $hasLesserCopyLeft(requirementsB)$ **then**
  if $LizennameA \neq LizennameB$ **then**
    if *connectionType.isDerivative* **then**
      $conflict \leftarrow true$
    end if
  end if
end if

```
if hasShareAlike(requirementsA) and hasShareAlike(requirementsB) then
    if LizenznameA ≠ LizenznameB then
        if connectionType.isDerivative then
            conflict ← true
        end if
    end if
end if

copyLeft ← hasCopyLeft(requirementsA)
isSource ← true

for each Requirement in requirementsA do
    if !requirementsB.contains(req) then
        if !req.compatibilityIfUnspecified(conType, isSource, copyLeft) then
            conflict ← true
        end if
    end if
end for

isSource ← false
for each Requirement in requirementsB do
    if !requirementsA.contains(req) then
        if !req.compatibilityIfUnspecified(conType, isSource, copyLeft) then
            conflict ← true
        end if
    end if
end for
```

Es gibt ein paar Annahmen, die für den Algorithmus getroffen wurden. Die Version der Lizenzen wird hier mit der Annahme überprüft, dass eine Lizenz nicht abwärtskompatibel ist. Das bedeutet, die Lizenzen werden so modelliert, dass die Version nur explizit mit angegeben wird, wenn keine Abwärtskompatibilität besteht.
Des Weiteren wird davon ausgegangen, dass *Lizenz A* immer Quelle und *Lizenz B* immer Ziel, bezogen auf den Verbindungstyp, ist.
Bei den Verboten und Genehmigungen wird davon ausgegangen, dass beide Lizenzen die gleichen haben müssen, da es sonst zu einem Konflikt kommt.
Für die Bedingungen bedarf es eines genaueren Vergleichs, im Besonderen wenn in einer Lizenz bestimmte Bedingungen vorhanden sind und in der anderen nicht. Das Verhalten für eine Bedingung, wenn diese nicht vorhanden ist, ist in den jeweiligen Bedingungen selbst definiert. Dies ist im Algorithmus durch die Methode *compatibilityIfUnspecified* dargestellt. Jene bekommt als Parameter den Verbindungstyp, sowie ein boolesches Argument zur Unterscheidung, ob es sich um

Quell- oder Zielkomponente handelt, für welche die Bedingung definiert ist. Dies ist wichtig, da die Richtung entscheidend für die Kompatibilität ist. Sollte z.B. für die Quelle *Copyleft* definiert sein und für das Ziel nicht, so sind diese Lizenzen kompatibel, ist es jedoch umgekehrt und für das Ziel ist Copyleft definiert und für die Quelle nicht und es handelt sich um ein Derivat oder eine kombinierte Arbeit, bedeutet dies, dass die Quelle die Komponente des Ziels als Derivat oder kombinierte Arbeit beinhaltet und deshalb wegen der Copyleft Klausel ein Konflikt entsteht.

Erweitertes Modell

In Abschnitt 2.3.6 wurden bereits die Überlegungen für die Funktionsweise des Lizenzvergleiches beschrieben. In diesem Abschnitt wird nun die Implementierung des erweiterten Ansatzes erläutert.
Um einen automatischen Vergleich von Lizenzen durchzuführen, mussten zunächst die in XML modellierten Lizenzen und Operationen in Java Klassen überführt werden.
Eine Lizenz enthält in Java eine Liste an Lizenzklauseln, zugeordnet zu ihren Operationen für welche sie gelten, analog zu der bereits beschriebenen XML Struktur. Die Lizenzklauseln *LicenseClauses* enthalten zusätzlich zu den Rechten *rights* und Pflichten *obligations* noch eine weitere Liste, welche die Pflichten beinhaltet, aus denen eine Auswahl getroffen werden kann (*selections*). Folgendes Klassendiagramm zeigt diese Struktur:

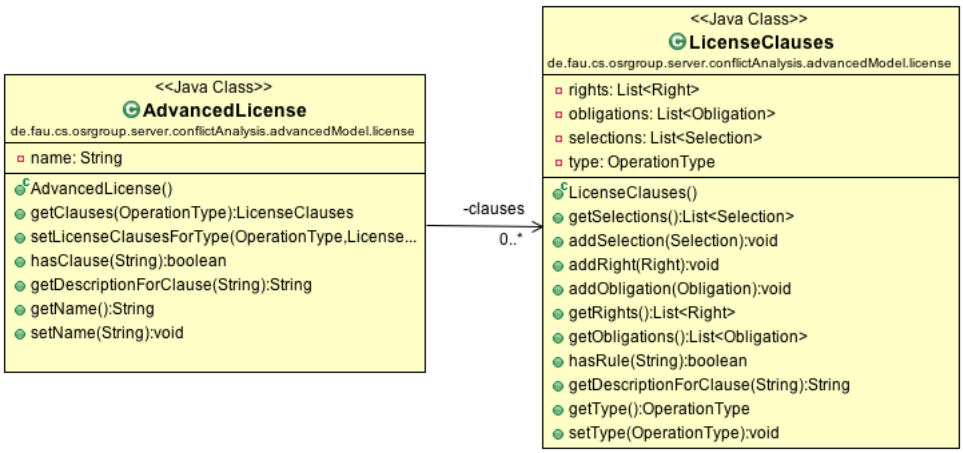


Abbildung 3.7: Klassendiagramm für Lizenzen

Die Operationen wurden in XML als rekursive Baumstruktur modelliert. Diese Struktur wurde in Java übernommen und sieht folgendermaßen aus:

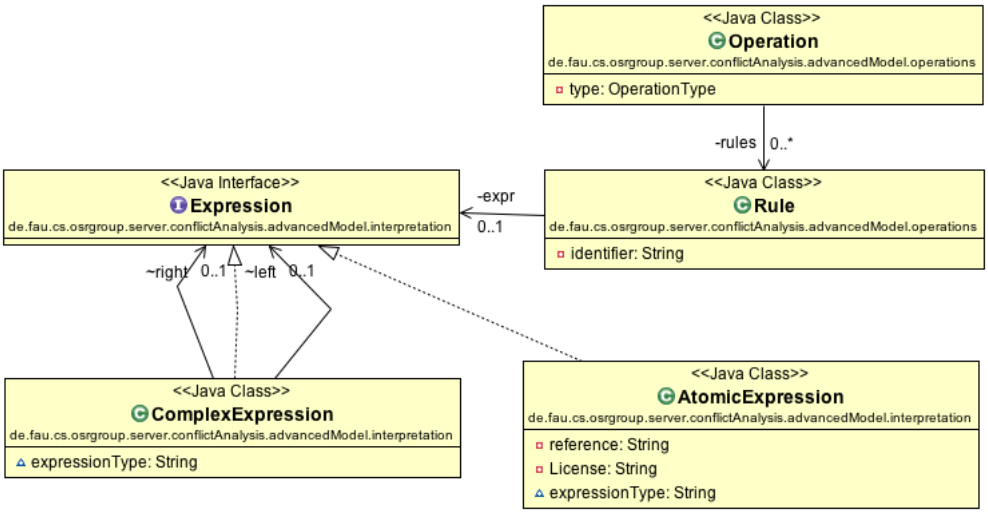


Abbildung 3.8: Klassendiagramm für Regeln und Ausdrücke

Es gibt eine Klasse *Operation*, welche eine Liste an *Rule* Objekten beinhaltet. Diese *Rule* Objekte besitzen immer eine Regel. Diese Regeln sind logische Ausdrücke und können zwei Formen annehmen. *AtomicExpression* steht dabei für einen Ausdruck, der keinen Nachfolger hat und direkt interpretiert werden kann. *ComplexExpression* hingegen hat eine linke und eine rechte Seite, welche wiederum einen Ausdruck (*Expression*) beinhalten. Auf diese Weise werden die Ausdrücke als eine Art abstrakter Syntaxbaum aufgebaut.
Nachdem nun alle nötigen Modelle in Java abgebildet sind, können die Lizenzen automatisch verglichen werden. Dafür müssen die Regeln interpretiert werden, die den Rechten und Pflichten der Lizenz für den gegebenen Verbindungstyp zugeordnet sind.
Diese Auswertung der logischen Ausdrücke wurde mit Hilfe des Visitor Pattern (Gamma, Helm, Johnson & Vlissides, 1995, S. 331-344) realisiert. Hierfür wurde ein *EvaluationVisitor* implementiert, der die Auswertung von atomaren und komplexen Ausdrücken durchführen kann. Da es für die komplexen Ausdrücke verschiedene Operatoren gibt, wurde deren Verhalten in extra Klassen gekapselt. Der Zugriff auf diese Klassen erfolgt über eine Factory Method (Gamma et al., 1995, S. 107-116), welche auf Basis des *expressionType* den richtigen Operator zurückgibt.
Auf diese Weise werden alle Regeln nacheinander überprüft und zum Schluss das Gesamtergebnis erstellt. Das Gesamtergebnis erhält als Wert *wahr*, wenn alle Regeln erfüllt sind, und *falsch*, sobald eine Regel als falsch interpretiert wurde.
Die Regeln, welche als falsch interpretiert wurden, werden dem Client übergeben, sodass dem Nutzer angezeigt werden kann, durch welche Rechte oder Pflichten ein Konflikt enstanden ist.

3.4 Gesamtarchitektur und Ablauf der Analyse

Nachdem nun die einzelnen Komponenten erläutert wurden, soll in diesem Abschnitt das Zusammenspiel erklärt werden. Die Gesamtarchitektur setzt sich aus den folgenden Bestandteilen zusammen. Es gibt einen Client mit einer Visualisierung und der beschriebenen Benutzerschnittstelle und einen Server, der die Komponenten zur Analyse der Softwarearchitektur enthält. Dazu zählen die Lizenzextraktion und die Lizenzanalyse sowieso Pakete zum Einlesen und Schreiben von JSON, RDF, XML und der eigens definierten ADL. Des Weiteren gibt es noch die Komponente, welche den FOSSology Server über REST an den Server anbindet. Im folgenden Diagramm ist diese Gesamtstruktur graphisch dargestellt:

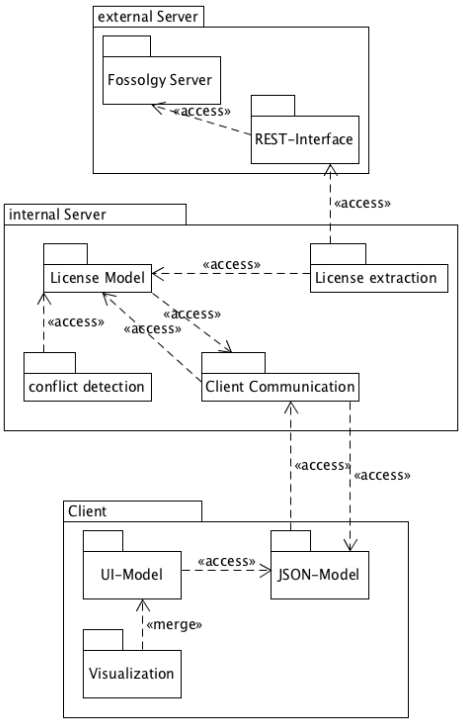


Abbildung 3.9: Gesamtarchitektur der Anwendung

Um den Ablauf der Modellierung und Analyse einer Softwarearchitektur besser zu veranschaulichen, wird hier das Ablaufdiagramm 3.10 vorgestellt.
Dieses Diagramm zeigt, wie die Nutzung der Anwendung abläuft. Es wird dargestellt, wie der Nutzer eine selbst definierte Architektur hochladen, diese editieren oder erweitern und analysieren kann. Beim Editieren gibt es die Möglichkeit, eine Datei für die Komponente hochzuladen, diese wird dann über die REST Schnittstelle zu dem FOSSology Server geladen. Sobald der Nutzer die Analyse

der Architektur startet, wird die aktuelle Architektur im JSON Format an den Server geschickt und in das interne Modell überführt. Nachdem dies geschehen ist, wird für jede Komponente des Modells die Lizenz bestimmt. Dies geschieht entweder indem die Lizenz im Modell bereits vorher festgelegt wurde oder durch die Analyse des Source Codes durch FOSSology. Nachdem alle Lizenzen bestimmt wurden, wird die Konfliktanalyse der Architektur gestartet. Diese erfolgt nach einem der beiden Algorithmen aus Abschnitt 3.3, je nachdem welchen der Nutzer gewählt hat. Dabei wird der gewählte Algorithmus auf alle Verbindungen angewandt und das Ergebnis für jede Verbindung zurückgegeben. In der Visualisierung wird dieses dann auf zwei Arten dargestellt. Für einen ersten Überblick werden die Verbindungen der Architektur in grün oder rot gefärbt. Grün steht dabei für "kein Konflikt" und rot für "Konflikt". Des Weiteren können genauere Informationen über die Konflikte in Erfahrung gebracht werden, indem der Nutzer mit der Maus über eine der roten Verbindungen geht. In diesem Fall erscheint ein Fenster, welches die Konflikte beschreibt.

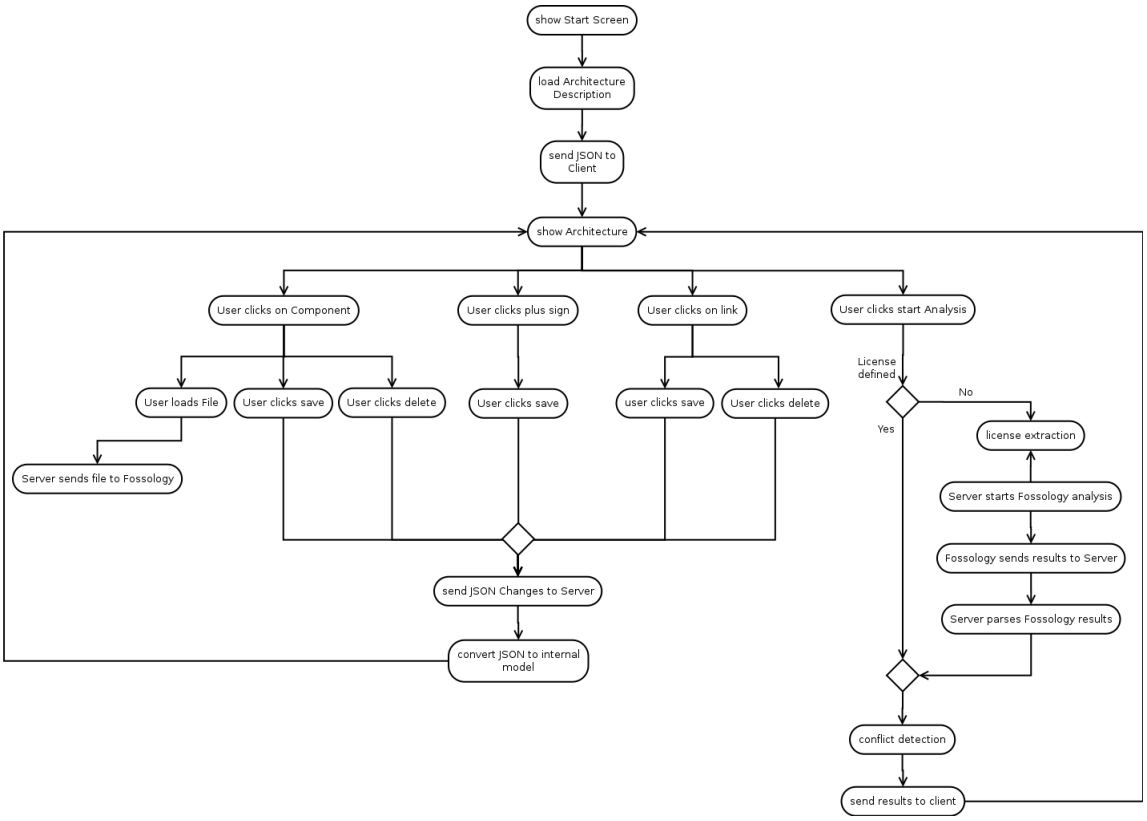


Abbildung 3.10: Ablaufdiagramm der Anwendung

4 Verifikation der Lizenzalgorithmen

Im folgenden Kapitel sollen die in dieser Arbeit entwickelten Lizenzmodelle, und die für die Modelle entwickelten Algorithmen mit Hilfe zweier Beispiele getestet, und deren Funktionalität veranschaulicht werden.
Ziel ist es, für ein reales Projekt und ein ausgedachtes Beispielprojekt die entsprechende Architektur zu modellieren und zu analysieren. Zunächst musste ein passendes Beispielprojekt konstruiert werden, welches sich gut eignet, um die verschiedenen Konzepte aus den Modellen darzustellen. Anschließend wurde die Architektur für dieses Projekt mit Hilfe der in dieser Arbeit entwickelten ADL modelliert.
Als zweiten Schritt wurde ein reales Projekt ausgewählt, welches auf seine Lizenzkompatibilität untersucht werden sollte. Hierfür war eine ausführliche Analyse nötig, um all die Subkomponenten zu extrahieren und die Beziehungen zwischen diesen zu bestimmen. Eine detailliertere Beschreibung zu diesen Schritten findet sich in Abschnitt 4.2.
Anschließend erfolgt jeweils eine manuelle und eine automatische Analyse der Architektur. Bei dieser Analyse geht es darum festzustellen, ob die vorliegende Architektur Lizenzkonflikte aufweist oder nicht. Für die automatische Analyse wurden die zwei entwickelten Algorithmen verwendet.
Zum Schluss werden dann die manuellen Ergebnisse mit denen der automatischen Analyse verglichen und es wird auf mögliche Unterschiede eingegangen.
Des Weiteren wird diskutiert, wo die Schwächen der beiden Modelle liegen und wie diese noch weiter verbessert werden könnten.

4.1 Modellierung einer Beispielkonfiguration

In dieser Arbeit wurden zwei verschiedene Möglichkeiten umgesetzt, eine Architektur auf Lizenzkonflikte zu untersuchen. Beide wurden wie beschrieben als Softwaremodul implementiert und sollen nun anhand eines Beispiels getestet wer-

den. Hierfür ist es wichtig, dass ein Beispiel konstruiert wird, dass die Stärken und Schwächen der Modelle aufzeigt. Es soll aufzeigen, dass Konflikte erkannt werden können und wie diese in dem Editor dargestellt werden.
Als Erstes wurde überlegt, welche Lizenz die Gesamtkomponente für dieses Beispiel haben soll. Die Entscheidung hierbei fiel auf die Apache License Version 2, da diese nicht so restriktiv ist wie die GPL Familie, aber auch nicht so tolerant wie z.B. die BSD oder MIT Lizenz.
Um das Zusammenspiel der Apache Lizenz mit den GPL-Lizenzen zu zeigen, wurden diese als Subkomponenten von der Gesamtkomponente modelliert. Des Weiteren soll mit diesem Beispiel auch aufgezeigt werden, ob die GPLv3 mit der Artistic License und der *Microsoft Public License* (MS-PL) License zusammen verwendet werden kann. Um das Zusammenspiel zwischen Open Source Lizenzen und Proprietären Lizenzen zu zeigen, wurde in diesem Beispiel die Corel Transactional License mit der LGPLv3 kombiniert.
Die gesamte Beispielarchitektur mit allen Komponenten und deren Lizenzen sind in der folgender Tabelle zu sehen:

Komponente	Lizenz
Gesamtkomponente	Apache-2.0
Subkomponente 1	LGPL-3.0
Subkomponente 2	GPL-3.0
Subkomponente 3	GPL-3.0
Subkomponente 4	BSD
Subkomponente 5	EPL
Subkomponente 6	CTL
Subkomponente 7	GPL-2.0
Subkomponente 8	Apache-2.0
Subkomponente 9	MS-PL
Subkomponente 10	Artistic
Subkomponente 11	MIT

Tabelle 4.1: Überblick der Komponenten mit zugehörigen Lizenzen

Die Verbindungen zwischen den einzelnen Komponenten wurden in folgender Tabelle dargestellt:

Komponente A	Komponente B	Verbindungstyp
Gesamtkomponente	Subkomponente 1	Kombination
Gesamtkomponente	Subkomponente 2	Nutzung
Gesamtkomponente	Subkomponente 3	Kombination
Gesamtkomponente	Subkomponente 4	Kombination
Subkomponente 1	Subkomponente 5	Kombination
Subkomponente 2	Subkomponente 6	Derivat
Subkomponente 2	Subkomponente 7	Derivat
Subkomponente 3	Subkomponente 9	Derivat
Subkomponente 3	Subkomponente 10	Derivat
Subkomponente 4	Subkomponente 11	Derivat

Tabelle 4.2: Verbindungen zwischen den Komponenten

Nachdem nun die einzelnen Komponenten, ihre Lizenzen und die Verbindungen untereinander definiert wurden, wurde die beschriebene Beispielarchitektur mittels der ADL modelliert. Das Ergebnis dieser Modellierung ist in folgender Abbildung zu sehen:

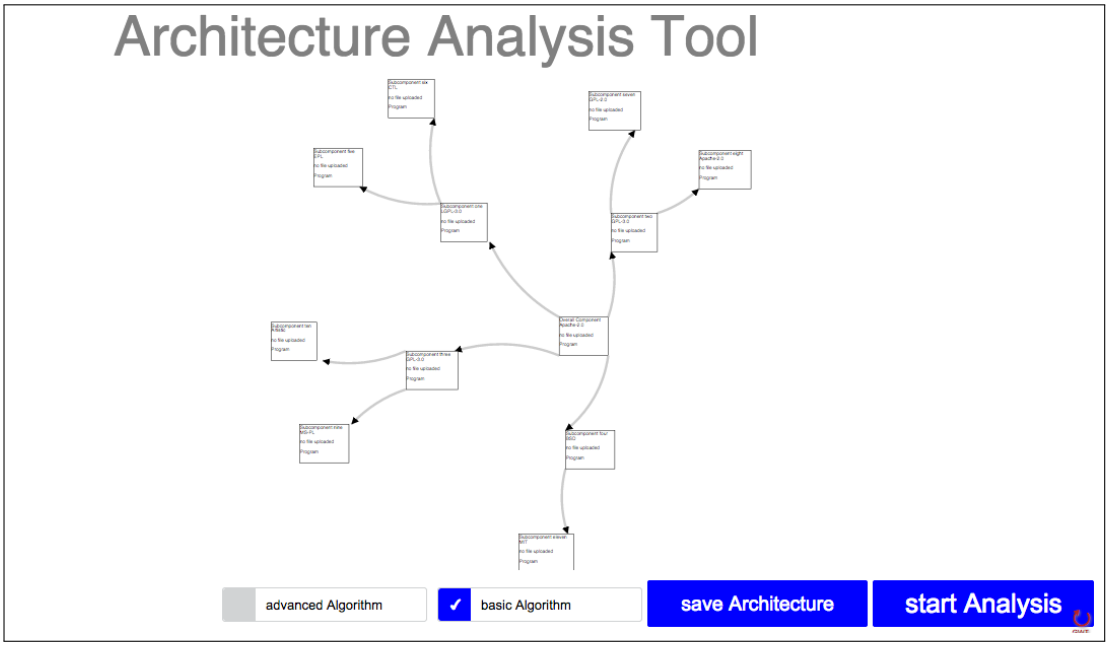


Abbildung 4.1: Ansicht der Beispielarchitektur im Editor

4.2 Modellierung von MediaWiki

Neben der Beispielkonfiguration, die dafür gedacht ist, die Stärken und Schwächen der beiden Algorithmen aufzuzeigen, soll hier noch zusätzlich ein reales Beispiel modelliert werden. Für diesen Fall wurde *MediaWiki*[1] ausgewählt. MediaWiki ist eine Open Source Software, welche für Wikis eingesetzt wird. Das bekannteste Beispiel dafür ist Wikipedia.
Um MediaWiki mit all seinen Abhängigkeiten modellieren zu können, muss zunächst analysiert werden, welche Abhängigkeiten vorhanden sind und auf welche Weise diese mit dem System verbunden sind. Das Ergebnis dieser Analyse ist in folgender Tabelle zu sehen:

Softwarepaket	Verbindungsart	Lizenz
Datenbank	Nutzung	GPL
jquery	Integration	MIT
jquery tipsy	Derivat	MIT
jquery UI	Integration	MIT
json2	Integration	MIT
Moment.js	Integration	MIT
Parsoid	Integration	GPL
jsminplus	Integration	GPL
es5shim	Integration	MIT
jquery chosen	Integration	MIT
qunit	Integration	MIT
sinon.js	Integration	MIT
CLDRPluralRuleParser.js	Derivative	MIT
msgpack-php	Derivative	BSD

Tabelle 4.3: MediaWiki Komponenten mit Verbindungstyp und Lizenz

Bei dem Datenbank Softwarepaket wurde für dieses Beispiel die MySQL Datenbank ausgewählt, da diese die restriktivste Lizenz der verfügbaren Datenbanken hat. Andere Datenbanken wie die PostgreSQL Datenbank (BSD oder MIT) oder die SQLite Datenbank (Public Domain) haben wesentlich tolerantere Lizenzen. Nach dem gleichen Prinzip wurde auch verfahren, wenn Pakete mehrere Lizenzen erlauben.
Die vollständige Architektur, modelliert mittels der ADL, ist in folgender Abbildung zu sehen:

[1] https://www.mediawiki.org/wiki/MediaWiki/de

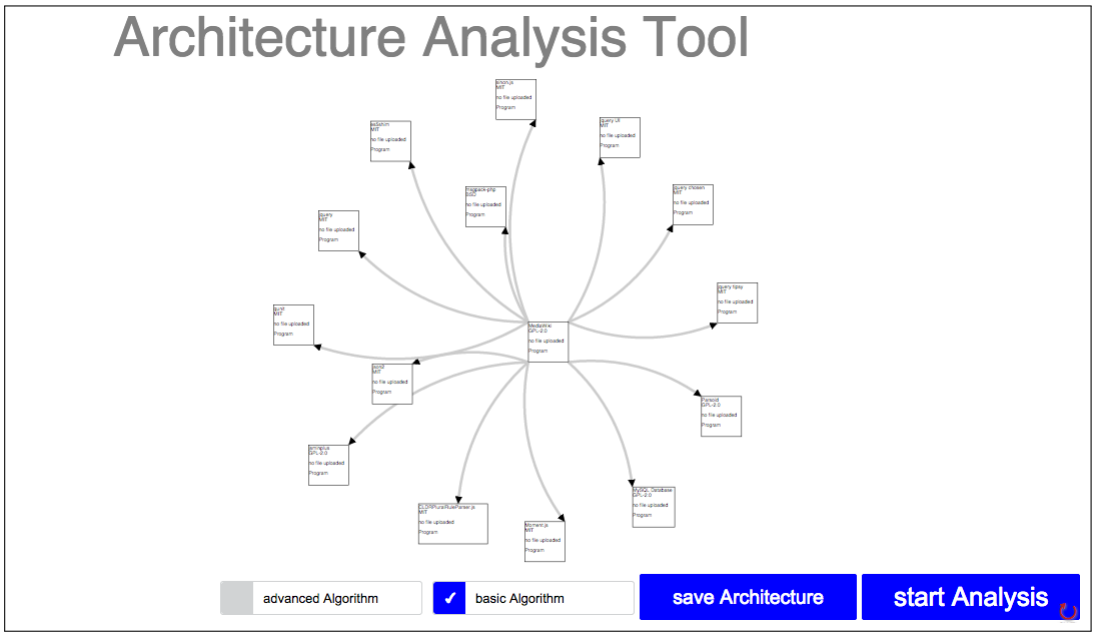


Abbildung 4.2: Ansicht der MediaWiki-Architektur im Editor

4.3 Manuelle und automatische Analyse

Zunächst sollen die beiden Architekturen in diesem Abschnitt manuell analysiert werden. Hierbei werden die vorhandenen Lizenzen und deren Kompatibilität untereinander, bezogen auf den Verbindungstyp, untersucht. Anschließend wird die gleiche Analyse mit Hilfe der beiden Softwarelösungen durchgeführt und im nächsten Abschnitt die Ergebnisse verglichen.

Manuelle Analyse der Beispielarchitektur
Bei der Beispielarchitektur wurde für die Gesamtarchitektur die Apache License Version 2 gewählt. Verbunden sind mit dieser Komponente noch vier weitere Komponenten.
Die erste Komponente ist über den Verindungstyp “Kombination” mit der Gesamtarchitektur verbunden und hat die Lizenz GNU Lesser General Public License (LGPLv3). Dies führt jedoch zu einem Konflikt, da die LGPLv3 die GPLv3 erweitert und diese unter Punkt 6b (*GNU General Public License Version 3*, 2015) vorschreibt, dass der Source Code immer mit verbreitet werden muss.
Die zweite Komponente besitzt die GPLv3 Lizenz und ist über den Verbindungstyp “Nutzung” mit der Gesamtarchitektur verbunden. Da hier nur eine reine Nutzung vorliegt, wie beispielsweise über eine Client-Server Verbindung, gibt es unter diesen Umständen keinen Konflikt. Anders ist dies jedoch bei der dritten

Komponente, die auch die GPLv3 als Lizenz besitzt, jedoch über den Verbindungstyp “Kombination” in das Gesamtsystem integriert ist. In diesem Fall liegt ein Konflikt vor, da eine Arbeit, die unter GPLv3 veröffentlicht wurde, laut Punkt 5c der GPLv3 (*GNU General Public License Version 3*, 2015) nur dann mit einer anderen Arbeit zu einem ganzen kombiniert werden kann, wenn dieses wiederum unter der GPL veröffentlicht wird. Die vierte Komponente hat die BSD 3-Clause Lizenz und ist mit dem Verbindungstyp “Kombination” mit der Gesamtarchitektur verbunden. Da die BSD 3-Clause keine Einschränkungen für die Integration in ein anderes System aufweist, liegt hier bei der Kombination mit der Apache License Version 2 auch kein Konflikt vor.
Die vier beschriebenen Komponenten, die mit der Gesamtarchitektur verbunden sind, haben selbst auch Subkomponenten. Die Subkomponente 1 mit der LGPLv3 hat zwei weitere Subkomponenten. Die erste Komponente, welche die *Eclipse Public License* (EPL) besitzt, ist über den Verbindungstyp “Kombination” mit der Subkomponente verbunden. In diesem Fall liegt ein Konflikt vor, da die EPL nach Punkt 3a (*Eclipse Public License - v 1.0*, 2015) vorschreibt, dass für die Kombination entweder die EPL oder eine kompatible Lizenz verwendet werden muss. Die zweite Subkomponente ist durch die CTL lizenziert. Da dies eine proprietäre Lizenz ist, welche durch den Verbindungstyp “Kombination” in die Komponente mit der LGPLv3 integriert ist, kommt es hier zu einem Konflikt. Der Grund dafür ist, dass die Komponente mit CTL nicht in eine Open Source Komponente integriert werden kann, da sonst Rechte und Pflichten wie z.B. die Verfügbarkeit des Source Codes verletzt werden.
Die Subkomponente 2 hat zwei weitere Subkomponenten. Eine davon hat die Lizenz GPLv2 und ist über die Verbindungsart “Kombination” verbunden. Da die Subkomponente 2 GPLv3 als Lizenz hat, führt dies zu einem Konflikt, da die GPLv2 und GPLv3 laut der Free Software Foundation nicht miteinander kompatibel sind. (FAQ about GNU Licenses, 2015) Das liegt unter anderen daran, dass in GPLv3 unter Punkt 11 (*GNU General Public License Version 3*, 2015) spezielle Klauseln für Patentfragen eingeführt wurden, die in GPLv2 nicht vorhanden sind. Die zweite Subkomponente hat die Apache License Version 2 mit dem Verbindungstyp “Derivative”. Diese ist laut der Apache Software Foundation in dieser Weise kompatibel mit der GPLv3. (*Apache License v2.0 and GPL Compatibility*, 2015)
Die Subkomponente 3 hat auch wiederum 2 Subkomponenten. Die erste von beiden hat MS-PL als Lizenz und steht mit der Subkomponente 3 über den Verbindungstyp “Derivat” in Verbindung. Da die Subkomponente 3 unter GPLv3 lizenziert ist und die MS-PL eine Copyleft Klausel besitzt, sind diese beiden miteinander inkompatibel. Die zweite Subkomponente hat die Artistic License 1.0, welche mit dem Verbindungstyp “Derivat” mit der Subkomponente verbunden ist. Bei der Kombination von GPL-3.0 und der Artistic License ist es unklar, ob diese kompatibel sind oder nicht. Die Subkomponente 4 hat eine weitere Subkomponente, welche unter der MIT-Lizenz steht. Da die Subkomponente 4 unter

der BSD 3-Clause steht und diese beiden Lizenzen es jeweils erlauben ein Derivat unter einer anderen Lizenz zu veröffentlichen, besteht hier kein Konflikt.
Durch dieses Beispiel kann man gut erkennen, dass ein großes Softwaresystem mit vielen unterschiedlich lizenzierten Komponenten durchaus mit großer Sorgfalt entworfen werden muss, da sonst leicht erhebliche lizenzrechtliche Probleme entstehen können.

Manuelle Analyse von MediaWiki
Bei der Modellierung von MediaWiki ist zu beachten, dass die Gesamtarchitektur mit der GPL-Lizenz lizenziert ist. Es gibt jedoch auch 14 verschiedene Subkomponenten die mit der Gesamtarchitektur in Verbindung stehen. Von diesen 14 Komponenten haben 9 die MIT-Lizenz, drei die GPL-Lizenz und eine die BSD-Lizenz. Von daher muss an dieser Stelle geklärt werden, ob die MIT-Lizenz und die BSD-Lizenz für die Verbindungstypen "Kombination" und "Derivat" mit der GPLv3 kompatibel sind. Da beide Lizenzen tolerante Lizenzen sind, die weder Copyleft Klauseln noch andere Einschränkungen bezüglich einer Lizenz für eine Kombination oder ein Derivat aufweisen, liegt auch kein Konflikt mit der GPL-Lizenz vor.

Automatische Analyse der Beispielarchitektur

Modell basierend auf ccREL
Auf die oben beschriebene Beispielarchitektur wird nun zunächst, der auf ccREL basierende Lizenzalgorithmus angewandt und das Ergebnis hier beschrieben.
Wie in Abbildung 4.3 zu sehen ist, hat der Algorithmus einige Konflikte in der Architektur gefunden.
Zunächst einmal wurde erkannt, dass bei der Integration einer LGPLv3 Komponente in eine Komponente unter der Apache License Version 2 ein Konflikt entsteht, da die Komponente mit der LGPLv3 Lizenz es verlangt, dass der Source Code zur Verfügung gestellt werden muss. Ein weiterer Konflikt besteht außerdem zwischen der Komponente mit der Apache License Version 2 und jener mit der GPLv3. Diese sind mittels dem Verbindungstyp "Kombination" miteinander verbunden und in diesem Fall schreibt GPLv3 vor, dass der Source Code zur Verfügung gestellt und eine kombinierte Arbeit, welche eine GPLv3 Komponente enthält, wiederum unter dieser Lizenz veröffentlicht werden muss. Auch die Integration der CTL in die Komponenten mit LGPL wurde als Konflikt erkannt. Der letzte Konflikt der von dem einfachen Lizenzalgorithmus erkannt wurde, ist zwischen der Komponente mit GPLv2 und der mit GPLv3, welche über den Verbindungstyp "Kombination" miteinander verbunden sind. Als Begründung für

den Konflikt wurden die unterschiedlichen Versionen der beiden Lizenzen genannt.
Alle weiteren Verbindungen aus der Beispielarchitektur wurden durch diesen Lizenzalgorithmus als konfliktfrei eingestuft. Das bedeutet, dass der Lizenzalgorithmus den Konflikt zwischen GPLv3 und MS-PL sowie den zwischen LGPLv3 und EPL nicht erkannt hat.

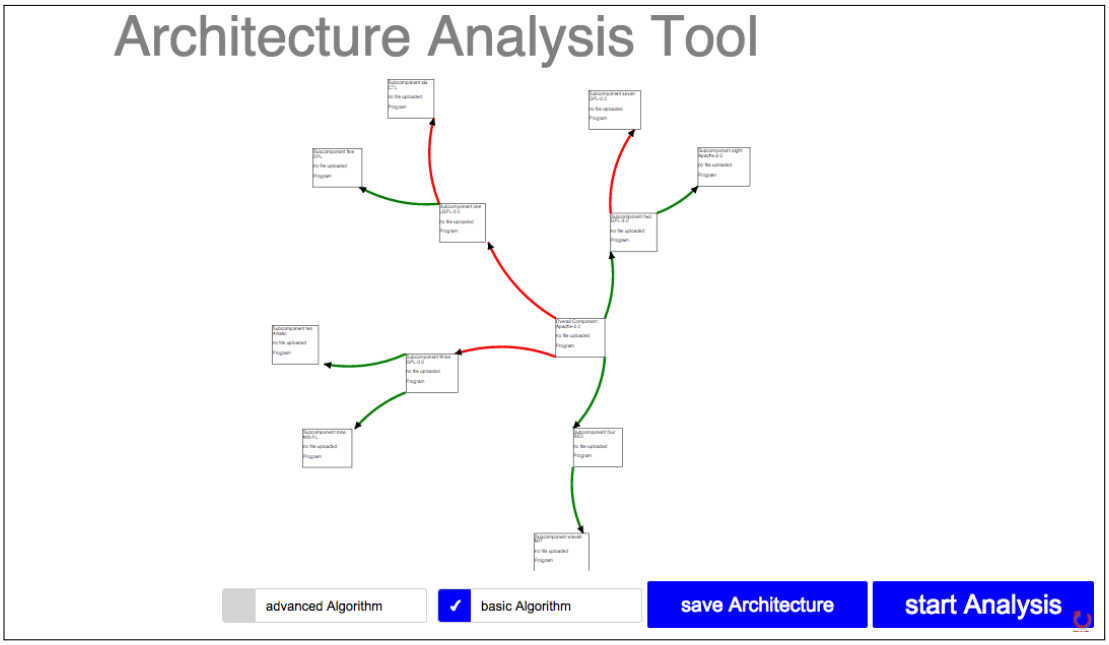


Abbildung 4.3: Ergebnis der Konfliktanalyse durch einfachen Lizenzalgorithmus für Beispielarchitektur

Erweitertes Modell

Im Folgenden wird das Ergebnis, welches durch die Anwendung des erweiterten Lizenzalgorithmus entstanden ist, besprochen. Wie in Abbildung 4.4 zu erkennen ist, hat der Algorithmus einige Lizenzkonflikte in der Beispielarchitektur aufgedeckt. Es wurde erkannt, dass eine Integration von einer Komponente unter der LGPLv3 in eine Komponente mit der Apache License Version 2 nicht erlaubt ist, da bei der LGPLv3 vorgeschrieben ist, dass der Source Code verfügbar gemacht werden muss. Des Weiteren wurde erkannt, dass eine Integration einer Komponente mit GPLv3 in eine Komponente mit Apache License Version 2 zu einem Konflikt führt. Die Gründe hierfür sind, dass der Source Code zur Verfügung stehen muss und dass für die Komponente die GPLv3, wegen der Copyleft Klausel, verwendet werden müsste. Ein weiterer Konflikt, der aufgedeckt wurde, ist, dass eine Komponente unter EPL nicht in eine Komponente mit Lizenz LGPLv3 integriert werden darf. Das liegt daran, dass hierfür entweder die EPL oder eine kompatible Lizenz verwendet werden müsste. Genauso wurde erkannt, dass die

Integration der CTL in die Komponente mit LGPL zu einem Konflikt führt.
Es wurde außerdem ein Konflikt zwischen der GPLv2 und der GPLv3 entdeckt. Dieser ist laut dem Lizenzalgorithmus darauf zurückzuführen, dass die Komponente, in welche der Code mit Lizenz GPLv2 integriert werden sollte, wiederum unter der gleichen Lizenz stehen muss.
Der letzte Konflikt, der entdeckt wurde, ist, dass ein Derivat mit der MS-PL Lizenz nicht in eine Komponente mit GPLv3 integriert werden kann.
Alle weiteren Verbindungen aus der Beispielarchitektur wurden durch den erweiterten Lizenzalgorithmus als kompatibel bewertet.

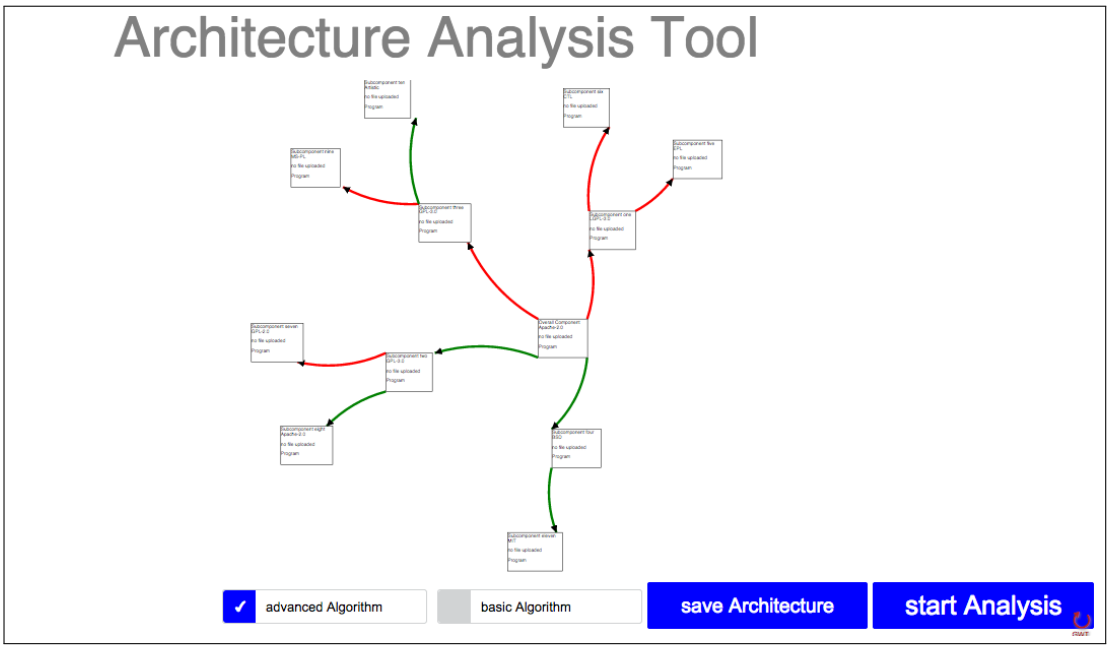


Abbildung 4.4: Ergebnis der Konfliktanalyse durch erweiterten Lizenzalgorithmus für Beispielarchitektur

Automatische Analyse von MediaWiki

Modell basierend auf ccREL

Für die automatische Analyse von MediaWiki wurde zunächst der Lizenzalgorithmus, basierend auf ccREL, verwendet und mit Hilfe dessen die Architektur auf Konflikte überprüft. Wie in Abbildung 4.5 anhand der grünen Verbindungslinien zu erkennen ist, liegt bei dieser Architektur laut dem einfachen Lizenzalgorithmus kein Konflikt vor.

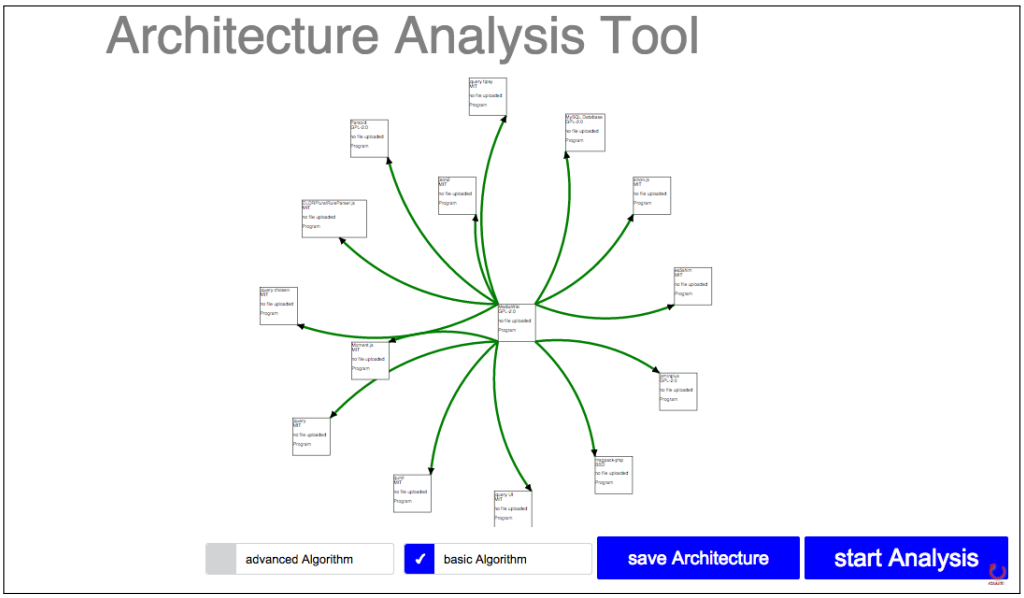


Abbildung 4.5: Ergebnis der Konfliktanalyse durch einfachen Lizenzalgorithmus für MediaWiki

Erweitertes Lizenzmodell

Wie in Abbildung 4.6 zu erkennen ist, wurden bei der Analyse von MediaWiki durch den erweiterten Lizenzalgorithmus auch keine Konflikte entdeckt.

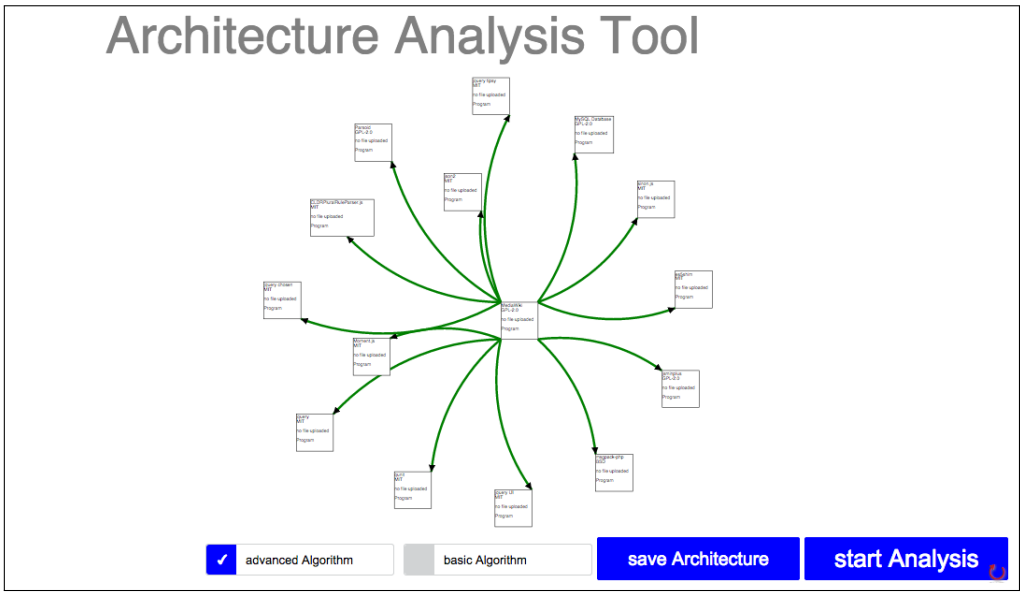


Abbildung 4.6: Ergebnis der Konfliktanalyse durch erweiterten Lizenzalgorithmus für MediaWiki

4.4 Vergleich der Ergebnisse

In diesem Abschnitt sollen die Ergebnisse der manuellen Analyse beider Beispiele mit den Ergebnissen der automatischen Analyse verglichen werden.

Vergleich für die Beispielarchitektur
Um die Ergebnisse übersichtlich darzustellen, kann zunächst der Tabelle 4.4 entnommen werden, zwischen welchen Verbindungen Konflikte bei der manuellen Analyse vorhanden sind und wie dies von den beiden Lizenzalgorithmen bewertet wird. Anschließend werden die wichtigsten Unterschiede diskutiert.

Verbindung	manuell	ccREL	erweitert
Apache 2.0 → LGPL-3.0: Kombination	✗	✗	✗
Apache 2.0 → GPL-3.0: Nutzung	✓	✓	✓
Apache 2.0 → GPL-3.0: Kombination	✗	✗	✗
Apache 2.0 → BSD: Kombination	✓	✓	✓
LGPL-3.0 → EPL: Kombination	✗	✓	✗
GPL-3.0 → GPL-2.0: Kombination	✗	✗	✗
GPL-3.0 → Apache 2.0: Derivat	✓	✓	✓
GPL-3.0 → MS-PL: Derivat	✗	✓	✗
GPL-3.0 → Artistic: Derivat	✓	✓	✓
BSD → MIT: Derivat	✓	✓	✓

Tabelle 4.4: Übersicht aller Ergebnisse für die Beispielarchitektur

Anhand dieser Tabelle ist zu erkennen, dass der erweiterte Lizenzalgorithmus dieselben Ergebnisse liefert wie die manuelle Analyse. Aber nicht nur die gleichen Konflikte wurden erkannt, es wurden auch dieselben Klauseln erkannt, welche zu diesen Konflikten führen. Beispielsweise wurde bei der Verbindung der Apache License Version 2 mit der LGPLv3 erkannt, dass hierbei ein Konflikt entsteht, weil die LGPLv3 vorschreibt, den Source Code immer zur Verfügung stellen zu müssen. Dieses Ergebnis zeigt, dass dieser Lizenzalgorithmus sehr gut auf die Problematik der Lizenzkonfliktanalyse angewandt werden kann und eine hohe Genauigkeit besitzt.
Bei dem Lizenzalgorithmus basierend auf ccREL ist zu erkennen, dass nicht alle Konflikte erkannt wurden. Zwischen der LGPLv3 und der EPL liegt für den Verbindungstyp "Kombination" nach der manuellen Analyse ein Konflikt vor. Dieser ist auf das Copyleft der EPL zurückzuführen, welches hier nicht erkannt wurde. Das gleiche Problem besteht bei der Analyse von GPLv3 mit MS-PL, welche mit dem Verbindungstyp "Derivat" verbunden sind. Hier wird auch das Copyleft

nicht richtig erkannt. Man erkennt daran, dass der einfache Lizenzalgorithmus etwas ungenauer arbeitet, jedoch trotz seiner geringen Anzahl an Requirements viele Fälle richtig erkennt.

Vergleich für MediaWiki
Bei MediaWiki ist der Vergleich der Ergebnisse einfach. Da in MediaWiki lediglich drei Lizenzen, die GPL, die MIT-Lizenz und die BSD-Lizenz verwendet wurden und die MIT und die BSD jeweils mit der GPL kompatibel sind, wurden hier keine Konflikte erkannt. Bei beiden Algorithmen konnte dies festgestellt werden.

5 Ergebnisse

Dieser Abschnitt soll die Ergebnisse der Arbeit zusammenfassen. Zunächst sollen die Schwierigkeiten und Probleme bei der Entwicklung sowie die Ergebnisse diskutiert werden. Anschließend wird ein Ausblick gegeben welcher aufzeigt, welche Möglichkeiten es gibt, die hier erarbeitete Lösung zu verbessern oder zu erweitern. Abschließend folgt eine Zusammenfassung und ein Fazit über den Ablauf der Arbeit.

5.1 Diskussion

Es war zunächst vorgesehen, FOSSology dafür zu verwenden, um die Softwarearchitektur eines gegebenen Projekts einzulesen und auf Basis von diesen Informationen die Darstellung der Architektur vorzunehmen. Es hat sich jedoch leider im Laufe der Arbeit herausgestellt, dass dies über FOSSology nicht realisierbar ist, da die Software keinerlei Möglichkeiten bietet die Informationen zur Architektur eines hochgeladenen Archivs abzufragen. FOSSology selbst führt auch keine Bestimmung der Komponentenbeziehungen durch, sondern überprüft lediglich die im Archiv enthaltenen Dateien separat auf ihre Lizenzen. Aus diesem Grund wurde dieser Ansatz wieder verworfen und auf die in der Arbeit vorgestellte manuelle Variante zurückgegriffen. Diese kann für ein gegebenes Archiv die Lizenzen zuverlässig extrahieren und die am häufigsten vorkommende Lizenz für die Komponente bestimmen.

Für das einfache Lizenzmodell wurden die Definitionen aus ccRel verwendet und um zwei Requirements erweitert. Im Vergleich zu dem erweiterten Lizenzmodell sind hier nur eine sehr begrenzte Zahl an Rechten und Pflichten, die allgemein gehalten wurden, vorhanden. Für die gängigsten Lizenzen hat die Analyse gezeigt, dass diese ausreichen, um einen Großteil der Konflikte zu erkennen. Problematischer wird es, sobald auch nicht geläufige Lizenzen mit spezielleren Klauseln integriert werden sollen. Hierfür müsste das gesamte Modell um weitere Rechte und Pflichten erweitert werden.

Für das erweiterte Lizenzmodell, wurden wie in Abschnitt 2.3.3 beschrieben, die häufigsten Lizenzen analysiert und die unterschiedlichen Klauseln extrahiert. Hier gilt es zu untersuchen, ob wirklich alle Klauseln notwendig sind oder ob eine weitere Zusammenfassung möglich ist. Eine Zusammenfassung hat den Vorteil, dass für eine Unterscheidung zwischen den Lizenzen nicht mehr so viele verschiedene Klauseln überprüft werden müssen. Sie birgt jedoch auch die Gefahr, dass wichtige Unterscheidungen durch eine falsche Zusammenfassung vernachlässigt und damit Konflikte übersehen werden. Des Weiteren wurden bisher nur die gängigsten Lizenzen untersucht und modelliert. Durch den Ansatz die Modellierung in XML auszulagern, kann jedoch die Menge an Klauseln beliebig erweitert werden und somit viele weitere Lizenzen einfach integriert werden. Für diese Integration müssen lediglich die Regeln für Derivative, Combination, Usage und General festgelegt werden. In der Implementierung selbst muss für eine Anpassung des Modells nichts geändert werden. Dies wäre lediglich der Fall wenn die momentan umgesetzten Operatoren für die Modellierung weiterer Lizenzen nicht ausreichen sollten. Dann müssten die neuen Operatoren ergänzt werden und der Algorithmus gegebenenfalls angepasst werden.

5.2 Ausblick

Außer der Erweiterung der Modelle um andere Lizenzen gibt es weitere Funktionalitäten, welche in Zukunft noch umgesetzt werden könnten.

Automatische Erkennung der Softwarearchitektur
Aktuell wird die Softwarearchitektur über eine in dieser Arbeit entwickelte ADL definiert und eingelesen. Obwohl dies gut funktioniert, und die Sprache sehr einfach anwendbar ist, ist es dennoch im Vorfeld immer notwendig, eine Modellierung der Architektur durchzuführen. Hier wäre es für die Zukunft sinnvoll, eine Lösung zu finden, die die Architektur aus dem Source Code heraus automatisch generiert. Arbeiten zu diesem Thema wurden in Abschnitt 2.2.2 vorgestellt.

Automatische Bestimmung der Lizenz
Bei der Bestimmung der Lizenz wird aktuell auf FOSSology zurückgegriffen. Hierbei werden die Lizenzen für die Dateien eines Archivs bestimmt und zurückgegeben. Gibt es mehrere Lizenzen für ein Archiv, wird die Lizenz, welche am häufigsten vorkommt, verwendet. Diese Vorgehensweise ist für die genaue Analyse der Subarchitektur des Archivs jedoch ungenügend. Es wäre wichtig, eine Lösung zu finden, welche den Aufbau der Komponente analysiert und bestimmen kann, wie die einzelnen Dateien mit unterschiedlichen Lizenzen zueinander in Beziehung stehen. Wenn diese Informationen vorhanden sind, kann auf die Subarchitektur

der Lizenzalgorithmus angewandt werden und so bestimmt werden, welche Lizenz für die Gesamtkomponente gilt.

Automatische Bestimmung der Verbindungstypen
Für eine bestehende Architektur werden aktuell die Verbindungstypen manuell mit angegeben. Dieser Vorgang erfordert eine genaue Analyse der Architektur und kann leicht zu Fehlern führen. Daher wäre es optimal wenn eine automatische Erkennung der Verbindungstypen vorhanden wäre. Hierfür könnten bereits bestehende Ansätze, wie beispielsweise der in Abschnitt 2.2.2 diskutierte, erweitert werden.

Erweiterung des Editors
Für den Editor wäre eine nächste Ausbaustufe, dem Nutzer es zu ermöglichen, Lizenzen selbst anzulegen und die Regeln, welche für den erweiterten Algorithmus notwendig sind, zu erweitern. Hierfür müsste ein möglichst einfacher und intuitiver Weg gefunden werden, so dass für den Nutzer keine aufwändige Einarbeitung notwendig ist.

Erweiterung der Lizenzalgorithmen

Lizenzalgorithmus basierend auf ccREL
Für den Lizenzalgorithmus basierend auf ccREL wurden die Requirements aus ccREL verwendet und erweitert. Wichtig wäre es, weitere Lizenzen mit diesem Modell zu modellieren, um zu überprüfen, ob die limitierte Anzahl an Requirements genügt. Sollte sich dabei herausstellen, dass die Anzahl zu gering ist und diese zu allgemein gehalten sind, müssten diese erweitert werden. In diesem Fall müsste untersucht werden, inwiefern sich dann die beiden Modelle noch unterscheiden.

Erweiterter Lizenzalgorithmus
Der erweiterte Lizenzalgorithmus wurde so gestaltet, dass möglichst einfach neue Lizenzen, und wenn nötig, neue Klauseln und zugehörige Regeln modelliert werden können. Um die Anwendung aus dieser Arbeit im realen Umfeld anwenden zu können, sollten noch weitere Lizenzen modelliert werden. Vor allem wären dabei Lizenzen mit nicht geläufigen Klauseln, interessant.
Ein weiterer Punkt, der in zukünftigen Arbeiten zu diesem Thema diskutiert werden könnte, ist, ob ein Mehrwert dadurch entsteht, dass Pflichten ihren Rechten zugeordnet werden und die Überprüfung entsprechend dieser Zuordnung erfolgt. In dieser Arbeit wurde darauf verzichtet, da im Endeffekt sowieso alle Rechte

und Pflichten überprüft werden müssen und aus der Zuordnung meiner Meinung nach kein Mehrwert entsteht.

5.3 Zusammenfassung

In der vorliegenden Arbeit wurde ein Editor, eine Architecture Description Language und zwei Lizenzmodelle entworfen, um komponentenbasierte Softwarearchitekturen modellieren und analysieren zu können. Der Schwerpunkt der Arbeit lag dabei auf der Konzeption der Lizenzmodelle und der Algorithmen zum Vergleich zwischen unterschiedlichen Lizenzen.
Ziel der Arbeit war es, einen Editor zu entwerfen, mit dessen Hilfe eine Softwarearchitektur dargestellt, bearbeitet und analysiert werden kann. Dabei sollte es möglich sein, eine Architektur vorab in einer Architecture Description Language zu modellieren und anschließend in den Editor zu laden. Diese Architektur kann dann erweitert und bearbeitet werden. Auf Basis der Verbindungs- und Lizenzinformationen, welche die Architektur enthält, sollte anschließend eine Analyse auf Lizenzkonflikte stattfinden. Diese Analyse verwendet, je nach Nutzerwahl, eines der beiden in dieser Arbeit entwickelten Lizenzmodelle und deren zugehörige Lizenzalgorithmen.
Als Erstes wurden im Forschungskapitel wichtige Grundlagen zu der Thematik dieser Arbeit erörtert. Anschließend wurden die wichtigsten Arbeiten, welche sich mit dem Vergleich von Lizenzen beschäftigt, vorgestellt und auf deren Stärken und Schwächen hingewiesen. Des Weiteren wurden Arbeiten besprochen, die sich mit der Analyse von Softwarearchitekturen sowie mit der Identifikation von Lizenzen auseinandersetzen.
Um eine Softwarearchitektur visuell darstellen zu können, muss dem Nutzer zunächst eine Möglichkeit zur Modellierung der Architektur gewährt werden. Für diesen Zweck wurde in der Arbeit eine einfache Architecture Description Language entwickelt. Diese ermöglicht die Erstellung einer Architektur als eine Menge von Komponenten und deren zugehörige Verbindungen. Die Komponenten enthalten dabei wichtige Metadaten, wie die Lizenz der Komponente, den Namen und die Art der Komponente und eine Beschreibung. Die Verbindungen enthalten Informationen darüber, um welchen Verbindungstyp es sich handelt und welche zwei Komponenten miteinander verbunden sind.
Der in GWT entwickelte Editor soll es dem Nutzer ermöglichen, diese Architekturbeschreibung zu laden und die beschriebenen Metadaten durch Auswahl der Komponente oder der Verbindung zu ändern. Des Weiteren ermöglicht dieser das Hinzufügen neuer Komponenten und Verbindungen, sowie das Entfernen bestehender. Für die Visualisierung wurde auf die gwt-d3 Bibliothek zurückgegriffen, welche einen Wrapper für die D3.js Bibliothek darstellt. Es wurde darauf geachtet, dass die Architektur als ein zyklischer Abhängigkeitsgraph dargestellt werden

kann, da es zwischen Softwarekomponenten auch bidirektionale Beziehungen geben kann.
Dem Nutzer soll es jedoch nicht nur möglich sein, eine Architektur zu modellieren, sich anzeigen zu lassen und zu bearbeiten, es soll auch eine Analyse der Lizenzen möglich sein, um Konflikte zwischen diesen erkennen zu können. Für die Bestimmung der Lizenzen wurden zwei Möglichkeiten erarbeitet. Die erste Möglichkeit besteht darin, dass der Nutzer die Lizenz bei der Modellierung der Architektur mit angibt. Die zweite Möglichkeit ist, dass der Nutzer den zugehörigen Source Code zu der Komponente im Editor hochlädt und anschließend die Lizenz automatisch bestimmt wird. Diese Bestimmung erfolgt über einen extern angebundenen FOSSology Server. Hierfür wurde eine REST-Schnittstelle entwickelt, welche den Server mit dem Editor verbindet. Über diese Schnittstelle werden die vom Nutzer bereitgestellten Dateien an Fossology weitergereicht. Fossology analysiert diese und gibt eine Liste an Lizenzen zurück. Die am häufigsten vorkommende wird dann als die Lizenz der Komponente angenommen.
Ein wesentlicher Bestandteil war es, die bestehenden Open Source Lizenzen zu analysieren und ein Lizenzmodell zu entwickeln. Hierfür wurden zwei Ansätze verfolgt und umgesetzt. Der erste baut auf ccRel auf und versucht mit Hilfe dieser Sprache eine Lizenz darzustellen. Diese Lizenzmodelle können dann durch einen dafür entwickelten Algorithmus miteinander verglichen werden und Aussagen über die Kompatibilität abgeleitet werden.
Der zweite Ansatz besteht darin, die häufigsten Open Source Lizenzen zu analysieren und die unterschiedlichen Rechte und Pflichten zu bestimmen. Da sich für die unterschiedlichen Verbindungstypen andere Rechte und Pflichten ergeben, wurden sie entsprechend aufgeteilt. Nun wurden noch Regeln mit Hilfe von logischen Ausdrücken festgelegt, die beschreiben, wie ein Vergleich für die einzelnen Rechte und Pflichten aussehen soll.
Die beiden Modelle wurden in den Editor integriert und dieser zeigt durch farbliche Kennzeichnung der Verbindungen (rot für Konflikt, grün für kompatibel) die Ergebnisse der Analyse an.
Zum Schluss der Arbeit wurden noch ein theoretisches und ein reales Beispiel modelliert und anhand dieser beiden die Funktionsweise der Lizenzmodelle und ihrer Lizenzalgorithmen aufgezeigt. Dabei hat sich gezeigt, dass beide Lizenzalgorithmen gute Ergebnisse liefern, und sich für die Analyse von komponentenbasierten Softwarearchitekturen gut eignen.

Anhang A Open Source Definition

Introduction Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

1. **Free Redistribution**
   The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

2. **Source Code**
   The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

3. **Derived Works**
   The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. **Integrity of The Author's Source Code**
   The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files"with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. **No Discrimination Against Persons or Groups**
   The license must not discriminate against any person or group of persons.

6. **No Discrimination Against Fields of Endeavor**
   The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. **Distribution of License**
   The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by

those parties.

8. **License Must Not Be Specific to a Product**
   The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. **License Must Not Restrict Other Software**
   The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

10. **License Must Be Technology-Neutral**
    No provision of the license may be predicated on any individual technology or style of interface.

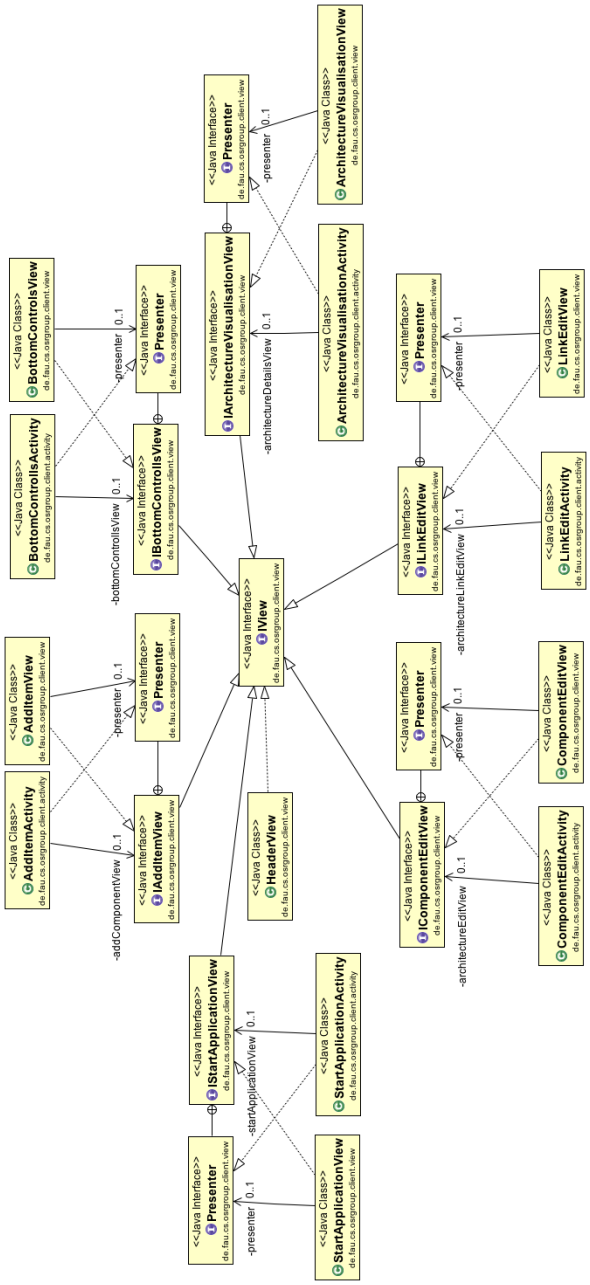
Anhang B Formale Darstellung GPLv3 (ccRel)

```
  <cc:License rdf:about="http://gnu.org/licenses/gpl-3.0.html">
    <dcq:hasVersion>3.0</dcq:hasVersion>
    <foaf:logo rdf:resource="http://www.gnu.org/graphics/gplv3-88x31.png"/>
    <foaf:logo rdf:resource="http://www.gnu.org/graphics/gplv3-127x51.png"/>
    <dcq:identifier>GNU GPL</dcq:identifier>
    <cc:requires rdf:resource="http://creativecommons.org/ns#Copyleft"/>
    <cc:requires rdf:resource="http://creativecommons.org/ns#SourceCode"/>
    <cc:requires rdf:resource="http://creativecommons.org/ns#Notice"/>
    <cc:requires rdf:resource="http://creativecommons.org/ns#Patent"/>
    <cc:permits rdf:resource="http://creativecommons.org/ns#Distribution"/>
    <cc:permits rdf:resource="http://creativecommons.org/ns#Reproduction"/>
    <cc:permits rdf:resource="http://creativecommons.org/ns#DerivativeWorks"/>
    <cc:legalcode rdf:resource="http://gnu.org/licenses/gpl-3.0.html"/>
    <dcq:title>GNU General Public License</dcq:title>
    <dcq:creator rdf:resource="http://fsf.org/"/>
    <osr:staticLink>
        <rdf:Bag>
            <rdf:li rdf:resource="http://www.osrgroup.de/masterthesis#derivative"/>
        </rdf:Bag>
    </osr:staticLink>
    <osr:dynamicLink>
        <rdf:Bag>
            <rdf:li rdf:resource="http://www.osrgroup.de/masterthesis#combined"/>
        </rdf:Bag>
    </osr:dynamicLink>
    <osr:fork>
        <rdf:Bag>
        </rdf:Bag>
    </osr:fork>
    <osr:ipc>
        <rdf:Bag>
        </rdf:Bag>
    </osr:ipc>
    <osr:linkedPlugin>
        <rdf:Bag>
            <rdf:li rdf:resource="http://www.osrgroup.de/masterthesis#derivative"/>
        </rdf:Bag>
    </osr:linkedPlugin>
    <osr:forkPlugin>
    </osr:forkPlugin>
    <osr:subclass>
        <rdf:Bag>
            <rdf:li rdf:resource="http://www.osrgroup.de/masterthesis#derivative"/>
        </rdf:Bag>
    </osr:subclass>
  </cc:License>
</rdf:RDF>
```

Anhang C Klauseln für erweitertes Modell

```
<Right id="1">may produce and distribute work based on the Program in source form</Right>
<Right id="2">may produce and distribute work based on the Program in object form</Right>
<Right id="3">may copy and distribute work in source form</Right>
<Right id="4">may copy and distribute work in object form</Right>
<Right id="5">may sublicense work under different license</Right>
<Right id="6">may produce combined work</Right>
<Right id="7">may use work based on the library in combined library</Right>
<Right id="8">may use the Program</Right>
<Right id="9">may charge fee for copy</Right>
<Right id="10">may use work with patents</Right>
<Obligation id="11">must include copyright notice</Obligation>
<Obligation id="12">must include disclaimer of warranty</Obligation>
<Obligation id="13">must include copy of this license</Obligation>
<Obligation id="14">must include change notice</Obligation>
<Obligation id="15">must include attribution notice</Obligation>
<Obligation id="16">must include notice file</Obligation>
<Obligation id="17">must display legal notice</Obligation>
<Obligation id="18">must distribute derivative under same license</Obligation>
<Obligation id="20">must not use contributor or copyright holder for promotion</Obligation>
<Obligation id="21a">must make modifications available to copyright holder</Obligation>
<Obligation id="21b">must use in-house only</Obligation>
<Obligation id="21c">use different name for non-standard executables.</Obligation>
<Obligation id="22a">must distribute standard version and instructions where to get it.</Obligation>
<Obligation id="22b">include source code with modifications</Obligation>
<Obligation id="22c">must allow redistribution of modification under this or similar license</Obligation>
<Obligation id="22d">must use different name for non-standard executables and include them and the standard executable</Obligation>
<Obligation id="25">must retain patent notice</Obligation>
<Obligation id="26">must retain trademark notice</Obligation>
<Obligation id="27">include instructions how to get source</Obligation>
<Obligation id="28">include copy of this license and GPLv3</Obligation>
<Obligation id="29">include notice about library usage</Obligation>
<Obligation id="30">must include work based on library</Obligation>
<Obligation id="31">must use compliant license</Obligation>
<Obligation id="32">must-not seek remedy based on warranty or liability</Obligation>
<Obligation id="33">must not institute patent litigation</Obligation>
<Obligation id="34">must not use trademark</Obligation>
<Obligation id="35">must make source code available</Obligation>
<Obligation id="36">must use this license</Obligation>
<Obligation id="37">must not use contributors name, logo or trademark</Obligation>
<Obligation id="38">must distribute derivative under GPLv3</Obligation>
```

Anhang D Architektur Client



Anhang E Beispiel: JSON-Format


```
{
"nodes": [

  {
  "ComponentId": "Component1",
  "name": "First Component",
  "description": "this is the first component",
  "license": "Apache-2.0",
  "licenseType": "foss",
  "fileName": "null"
  "componentType": "Program"
  },
  {
  "ComponentId": "Component2",
  "name": "second Component",
  "description": "this is the second component",
  "license": "Apache-2.0",
  "licenseType": "foss",
  "fileName": "null"
  "componentType": "Program"
  },
  {
  "ComponentId": "Component3",
  "name": "third Component",
  "description": "this is the third component",
  "license": "GPL-3.0",
  "licenseType": "foss",
  "fileName": "null"
  "componentType": "Program"
}

],
"links": [

  {
  "id": "Connection1",
  "description": "connection one",
  "source": "Component1",
  "target": "Component2",
  "conflict": "unknown",
  "conflictDescription": null,
  "relationshipType": "derivat"
  "connectionType": "staticLink"
  "relationshipType": "Derivative"
  },
  {
  "id": "Connection2",
  "description": "connection two",
  "source": "Component1",
```

```
  "target": "Component3",
  "conflict": "unknown",
  "conflictDescription": null,
  "relationshipType": "derivat"
  "connectionType": "staticLink"
  "relationshipType": "Derivative"
}

]
}
```

Literaturverzeichnis

17 U.S.Code § 101. (2015). Zugriff am 10.02.2015 auf `http://www.law.cornell.edu/uscode/text/17/101`

Abelson, H., Adida, B., Linksvayer, M. & Yergler, N. (2008). *ccREL: The Creative Commons Rights Expression Language.*

Alspaugh, T. A., Asuncion, H. U. & Scacchi, W. (2009a). Intellectual Property Rights Requirements for Heterogeneously-Licensed Systems. In *17th Internnational Conference on Requirements Engineering (RE09)* (S. 24-33). IEEE Computer Society.

Alspaugh, T. A., Asuncion, H. U. & Scacchi, W. (2009b). Software Licenses, Open Source Components, and Open Architectures. In *6th Acquisition Research Symposium* (S. 258–274).

Alspaugh, T. A., Scacchi, W. & Asuncion, H. U. (2010). Software Licenses in Context: The Challenge of Heterogeneously-Licensed Systems. *Journal of the Association for Information Systems*, 730–755.

Apache License v2.0 and GPL Compatibility. (2015). Zugriff am 09.03.2015 auf `https://www.apache.org/licenses/GPL-compatibility.html`

Breaux, T. D., Antón, A. I. & Doyle, J. (2008). Semantic Parameterization: A Process for Modeling Domain Descriptions. *ACM Trans. Softw. Eng. Methodol.*, 5:1–5:27.

cp2foss. (2015). Zugriff am 29.01.2015 auf `http://www.fossology.org/projects/fossology/wiki/Cp2foss`

Describing Copyright in RDF. (2015). Zugriff am 22.03.2015 auf `https://creativecommons.org/ns#`

DiBona, C., Ockman, S. & Stone, M. (1999). *Open Sources: Voices from the Open Source Revolution.* O'Reilly & Associates, Inc.

Eclipse Public License - v 1.0. (2015). Zugriff am 09.03.2015 auf `https://www.eclipse.org/legal/epl-v10.html`

fo_nomos_license_list. (2015). Zugriff am 29.01.2015 auf http://www.fossology.org/projects/fossology/wiki/Fo_nomos_license_list

fossjobs. (2015). Zugriff am 29.01.2015 auf http://www.fossology.org/projects/fossology/wiki/Fossjobs

Free Software Foundation. (2015). Zugriff am 27.01.2015 auf https://www.fsf.org/about/

Frequently Asked Questions about the GNU Licenses. (2015). Zugriff am 09.03.2015 auf https://www.gnu.org/licenses/gpl-faq.html#v2v3Compatibility

The Future of Open Source. (2014). Zugriff am 27.01.2015 auf http://www.northbridge.com/2014-future-open-source-survey-results-0

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley.

Germán, D. M., González-Barahona, J. M. & Robles, G. (2007). A Model to Understand the Building and Running Inter-Dependencies of Software. In *WCRE* (S. 140-149). IEEE Computer Society.

German, D. M. & Hassan, A. E. (2009). License Integration Patterns: Addressing License Mismatches in Component-based Development. In *Proceedings of the 31st International Conference on Software Engineering* (S. 188–198). IEEE Computer Society.

GNU General Public License Version 3. (2015). Zugriff am 09.03.2015 auf https://www.gnu.org/copyleft/gpl.html

The GNU Project. (2014). Zugriff am 27.01.2015 auf https://www.gnu.org/gnu/thegnuproject.de.html

Gobeille, R. (2008). The FOSSology Project. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (S. 47-50). ACM.

Google Web Toolkit. (2015). Zugriff am 30.01.2015 auf http://www.gwtproject.org/

Laurent, A. M. S. (2004). *Understanding Open Source and Free Software Licensing.* O'Reilly Media, Inc.

Linksvayer, M. (2009). *Free Software Foundation introduces RDF for GNU licenses.* Zugriff am 30.01.2015 auf http://creativecommons.org/weblog/entry/15406

MVP. (2015). Zugriff am 30.01.2015 auf http://www.gwtproject.org/articles/testing_methodologies_using_gwt.html

Penta, M. D., Germán, D. M. & Antoniol, G. (2010). Identifying licensing of jar archives using a code-search approach. In *MSR* (S. 151-160). IEEE Computer Society.

Ramsdale, C. (2010). *Building MVP apps.* Zugriff am 10.02.2015 auf `http://www.gwtproject.org/articles/mvp-architecture.html`

Rosen, L. (2004). *Open Source Licensing.* Prentice Hall.

Seriai, A.-D. & Chardigny, S. (2011). A Genetic Approach for Software Architecture Recovery from Object-Oriented Code. In *SEKE* (S. 515-520). Knowledge Systems Institute Graduate School.

UiBinder. (2009). Zugriff am 30.01.2015 auf `http://www.gwtproject.org/doc/latest/DevGuideUiBinder.html#Overview`

Villata, S. & Gandon, F. (2012). Licenses Compatibility and Composition in the Web of Data. In *COLD.* CEUR-WS.org.

Vogel, O., Arnold, I., Chughtai, A., Edmund, I., Timo, K., Mehlig, U. & Zdun, U. (2009). *Software-Architektur : Grundlagen - Konzepte - Praxis.* Spektrum Akademischer Verlag.

xArch: extensible XML-based representation for software architectures. (2015). Zugriff am 23.01.2015 auf `http://isr.uci.edu/projects/xarchuci/`