Dependency Management 2.0 – A Semantic Web Enabled Approach

Ellis Emmanuel Eghan

A Thesis
In the Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montreal, Quebec, Canada

July 2019

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Ellis Emmanuel Eghan**

Entitled: **Dependency Management 2.0 – A Semantic Web Enabled Approach**

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

Doctor of Philosophy (Computer Science)

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

Signed by the final examining committee:

______________________________	Chair
Dr. William Lynch	
______________________________	External Examiner
Dr. Giuliano Antoniol	
______________________________	Examiner
Dr. Ferhat Khendek	
______________________________	Examiner
Dr. Dhrubajyoti Goswami	
______________________________	Examiner
Dr. Nikolaos Tsantalis	
______________________________	Supervisor
Dr. Juergen Rilling	

Approved by ______________________________
Dr. Volker Haarslev, Graduate Program Director

3 September 2019 ______________________________
Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Dependency Management 2.0 – A Semantic Web Enabled Approach

Ellis Emmanuel Eghan, Ph.D.

Concordia University, 2019

Software development and evolution are highly distributed processes that involve a multitude of supporting tools and resources. Application programming interfaces are commonly used by software developers to reduce development cost and complexity by reusing code developed by third-parties or published by the open source community. However, these application programming interfaces have also introduced new challenges to the Software Engineering community (e.g., software vulnerabilities, API incompatibilities, and software license violations) that not only extend beyond the traditional boundaries of individual projects but also involve different software artifacts. As a result, there is the need for a technology-independent representation of software dependency semantics and the ability to seamlessly integrate this representation with knowledge from other software artifacts.

The Semantic Web and its supporting technology stack have been widely promoted to model, integrate, and support interoperability among heterogeneous data sources. This dissertation takes advantage of the Semantic Web and its enabling technology stack for knowledge modeling and integration. The thesis introduces five major contributions: (1) We present a formal Software Build System Ontology – SBSON, which captures concepts and properties for software build and dependency management systems. This formal knowledge representation allows us to take advantage of Semantic Web inference services forming the basis for a more flexibility API dependency analysis compared to traditional proprietary analysis approaches. (2) We conducted a user survey which involved 53 open source developers to allow us to gain insights on how actual developers manage API breaking changes. (3) We introduced a novel approach which integrates our SBSON model with knowledge about source code usage and changes within the Maven ecosystem to support API consumers and producers in managing (assessing and minimizing) the impacts of breaking changes. (4) A Security Vulnerability Analysis Framework

(SV-AF) is introduced, which integrates builds system, source code, versioning system, and vulnerability ontologies to trace and assess the impact of security vulnerabilities across project boundaries. (5) Finally, we introduce an Ontological Trustworthiness Assessment Model (OntTAM). OntTAM is an integration of our build, source code, vulnerability and license ontologies which supports a holistic analysis and assessment of quality attributes related to the trustworthiness of libraries and APIs in open source systems.

Several case studies are presented to illustrate the applicability and flexibility of our modelling approach, demonstrating that our knowledge modeling approach can seamlessly integrate and reuse knowledge extracted from existing build and dependency management systems with other existing heterogeneous data sources found in the software engineering domain. As part of our case studies, we also demonstrate how this unified knowledge model can enable new types of project dependency analysis.

To Betty, Michael, and Isabel

ACKNOWLEDGEMENTS

First and foremost, all praises to God for blessing, protecting, and guiding me throughout my studies. I could never have accomplished this without my faith.

A heartfelt thanks goes to my supervisor, Dr. Juergen Rilling, for having given me such an opportunity. Dr. Rilling did more than just advising: his continuous support, patience, motivation, and immense knowledge pushed me beyond my boundaries and shaped me, both as a person and a researcher. Throughout this journey, Dr. Rilling was always available to openly discuss new research ideas, ask challenging questions, and therefore elevate this work to the best it can be. His unique personality as a supervisor and friend is the main reason behind the success of this research.

I also extend my thanks to members of the examining committee including Dr. Nikolaos Tsantalis, Dr. Ferhat Khendek, Dr. Dhrubajyoti Goswami, and Dr. Giuliano Antoniol (External), whose work and valuable feedback expanded my horizon and helped me discover new research opportunities that significantly improved my research.

My gratitude also goes to Ambient Software Evolution Group members at Concordia University: Dr. Sultan Alqahtani, Dr. Rabe Abdalkareem, SayedHassan Khatoonabadi, Chris Forbes, Parisa Moslehi, and Yasaman Sarlati. A special thanks to two unique friends and colleagues, Dr. Sultan Alqahtani and Parisa Moslehi, who were there during the past years to discuss and share research ideas and challenges. Our discussions led to new published work and more research opportunities.

Above all, I thank all the members of my family. First and foremost, I want to thank my parents, Victor and Joyce Eghan, who generously and wholeheartedly gave me their unconditional love and endless support throughout these years. I am grateful for your trust and confidence in me and for giving me the freedom to pursue my dreams. Last, but definitely most prominently, I thank my wife, Betty, for her unwavering love and encouragement during the pursuit of my studies. Thank you for always believing in me, and for reminding me to endure during the tough times. I consider myself lucky to have encountered you. I love you.

Table of Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
BCD	Breaking Change Density
BCI	Breaking Change Impact
CVE	Common Vulnerabilities Exposure
CWE	Common Weakness Enumeration
DL	Description Logic
LOVC	Lines of Vulnerable Code
LVC	License Violation Count
MARKOS	The MARKet for Open Source license ontology
MSR	Mining Software Repositories
NVD	National Vulnerability Database
OMG	The Object Management Group
OntTAM	ONTology-based Trustworthiness Assessment Model
OSS	Open Source Software
OWA	Open World Assumption
OWASP	The Open Web Application Security Project
OWL	The Web Ontology Language
PSL	Probabilistic Soft Logic
RDF	Resource Description Framework
RDFS	The RDF Schema
SBSON	The Software Build System Ontology
SE	Software Engineering
SE-EQUAM	The Evolvable Quality Metamodel
SEON	The Software Engineering Ontologies
SEVONT	The Security Vulnerability Ontology
SOCON	The Source Code Ontology
SPARQL	A Simple Protocol and RDF Query Language
SV-AF	Security Vulnerability Analysis Framework
SW	The Semantic Web
SWRL	Semantic Web Rule Language
WVD	Weighted Vulnerability Density

Chapter 1

1 Introduction

Traditional software development processes, with their focus on closed architectures and platform-dependent software, restrict potential code reuse across project and organizational boundaries. With the introduction of the Internet, these restrictions have been removed, allowing for global access, online collaboration, information sharing, and internationalization of the software industry [1]. Software development and maintenance tasks can now be shared amongst team members working across and outside organizational boundaries. Code reuse through resources such as software libraries, components, services, design patterns, and frameworks published on the Internet has become an essential aspect allowing developers to reuse and share artifacts among developers and organizations. According to Mileva [2], "*most of today's software projects heavily depend on the usage of external libraries*." This use of libraries allows software developers to take advantage of features provided by Application Programming Interfaces (APIs) without having to reinvent the wheel [3], [4].

Automated dependency management environments have been introduced to further simplify the integration and reuse of external libraries during development. Developers no longer have to manually manage internal and external libraries their projects depend on. Build systems and dependency management tools automatically download and manage all required dependent components (including transitive dependencies), automatically update dependencies to their latest versions, and perform necessary dependency mediation (conflict resolution) when multiple versions of a dependency are encountered. Among the most commonly used open source build (dependency) repositories are Maven Central[1], npm[2], and RubyGems[3].

[1] https://search.maven.org/
[2] https://www.npmjs.com/
[3] https://rubygems.org/

Existing research has demonstrated how mining knowledge captured in these build repositories can be used to enhance software tasks such as identifying inconsistencies in license compliance [5], predicting build changes [6], [7], identifying build clones [8], and automatic library recommendation and migration [2].

Common to these approaches is that they use build and dependency repositories as information silos, which are not directly integrated and linked with other software repositories and therefore limiting their ability to share and reuse these analysis results for future analysis (both by humans and machines).

Furthermore, while existing software analysis and dependency approaches perform well in analyzing individual project contexts, the collaborative nature of today's software development requires new types of analysis and knowledge modeling approaches to address these global software engineering challenges. These challenges extend beyond the boundaries of individual projects due to dependency relationships among software projects and complete software ecosystems. There is the need for a technology-independent representation of software dependency semantics and the ability to seamlessly integrate such a representation with knowledge from other software artifacts.

In our research, we introduce a novel approach which takes advantage of the Semantic Web (SW) and its technology stack (e.g., ontologies, Linked Data, reasoning services) to establish a unified knowledge representation of build and dependency repositories. Based on this SW enabled representation, we can now further extend this knowledge base by integrating other (heterogeneous) resources to form the basis for a novel, flexible global impact analysis approach. Such a global impact analysis approach can provide both producers and consumers of software libraries with additional insights and guide them during the evolution of their libraries. Much of the flexibility of our approach is based on the use of inference services to reason upon knowledge that is explicit and implicit captured in the knowledge base.

1.1 Our Thesis

Despite the existing role of project dependency repositories and build system dependency management features, little is known on how this software dependency information can be integrated with other software-related knowledge to improve software development processes. This observation leads us to the formation of the following thesis:

A technology-independent representation of software dependency semantics, seamlessly integrated with other software artifacts, is needed to truly leverage project dependency information in software tasks.

To validate our thesis, we propose a knowledge modeling approach that supports the integration of heterogeneous knowledge resources such as software dependency, source code, vulnerability, and license information. On top of this knowledge model, we developed a set of applications that analyze the impact of code reuse through APIs, within a traditional project scope but also in a more global scope, across project boundaries.

1.2 Summary of Research Contributions

In this thesis, we make the following contributions:

- We conducted a survey involving 53 open source developers to gain insights on how they manage API breaking changes.
- Based on the survey results, we present a formal unified ontological model (SBSON, **S**oftware **B**uild **S**ystem **ON**tology) which captures concepts and properties for software build systems (Chapter 4). This formal knowledge representation allows us to take advantage of inference services provided by the SW, forming the basis for a more flexibility API dependency analysis compared to traditional proprietary analysis approaches.
- We introduced a novel approach to support API consumers and producers in managing (assessing and minimizing) the impacts of breaking changes. (Chapter 5). The main contributions of this approach are:
    - We use our knowledge model to identify the potential impact of breaking changes across project boundaries to support library consumers and producers in managing API breaking changes, by taking advantage of SW reasoning services.
    - We present a case study to demonstrate the applicability and flexibility of our approach in supporting library consumers while managing the impacts of breaking changes.

- We developed a Security Vulnerability Analysis Framework (SV-AF) to support evidence-based vulnerability detection (Chapter 6). The main contributions of this framework are:
    - Integration of different ontologies such as builds systems ontologies, source code ontologies, version systems ontologies, and vulnerabilities ontologies.
    - Applying ontologies alignment using Probabilistic Soft Logic (PSL) to establish weighted links between ontologies.
    - Performed case studies to illustrate the applicability of the presented approach in tracing and assessing the impact of security vulnerabilities across project boundaries.
- We introduce a novel Ontological Trustworthiness Assessment Model (OntTAM), an extension of the previous generic SE-EQUAM software assessment model [9] (Chapter 7). OntTAM is an integration of our build, source code, vulnerability and license ontologies which supports the automated analysis and assessment of quality attributes related to the trustworthiness of libraries and APIs in open source systems. The main contributions of this assessment model are:
    - We extend the MARKOS license ontology [10] with semantic rules for three categories of license violations.
    - We introduce new trustworthiness measures, which measure API breaking changes, security vulnerabilities, and license violations.
    - We perform several case studies to illustrate how our approach provides developers with additional insights on the potential impact of reused libraries and APIs on the quality and trustworthiness of their project.

A complete list of published works relevant to this dissertation can be found in the next section.

1.3 Related Publications

Earlier versions of the work completed in this thesis have been published in the following papers:

1- **E. E. Eghan**, S. S. Alqahtani, C. Forbes and J. Rilling, "API trustworthiness: an ontological approach for software library adoption," *Software Quality Journal,* 2019. https://doi.org/10.1007/s11219-018-9428-4.

2- S. S. Alqahtani, **E. E. Eghan** and J. Rilling, "Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach," *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, 2017, pp. 80-91.

3- S. S. Alqahtani, **E. E. Eghan** and J. Rilling, "SV-AF — A Security Vulnerability Analysis Framework," *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Ottawa, ON, 2016, pp. 219-229.

4- S. S. Alqahtani, **E. E. Eghan** and J. Rilling, "Tracing known security vulnerabilities in software repositories – A Semantic Web enabled modeling approach", *Science of Computer Programming*, Volume 121, 2016, pp. 153-175.

1.4 Thesis Organization

In what follows, we provide an overview of the thesis structure. Figure 1.1 summarizes the main sections of the thesis and their content, which are: In Chapter 2, we will discuss the motivation for the research presented in this thesis. Chapter 3 covers background and related work, including the SW technologies used for our knowledge model construction, mining software repositories (MSR), and dependency management with build systems. The chapter also covers existing works relevant to each of these topics. Chapter 4 describes the approach used to create our unified representation of build and dependency repositories. Chapters 5, 6 and 7 demonstrate how our unified model integrates knowledge from other software artifacts for flexible global software analysis. The conclusions and some promising avenues for future work are discussed in Chapter 8.

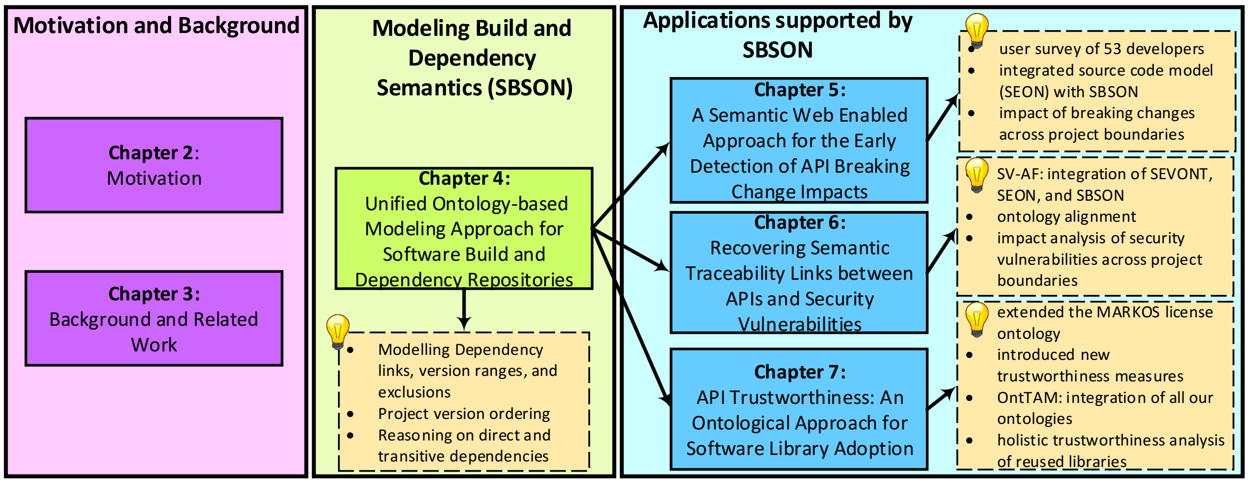

Motivation and Background
Chapter 2: Motivation
Chapter 3: Background and Related Work
Modeling Build and Dependency Semantics (SBSON)
Chapter 4: Unified Ontology-based Modeling Approach for Software Build and Dependency Repositories
Modelling Dependency links, version ranges, and exclusions
Project version ordering
Reasoning on direct and transitive dependencies
Applications supported by SBSON
Chapter 5: A Semantic Web Enabled Approach for the Early Detection of API Breaking Change Impacts
Chapter 6: Recovering Semantic Traceability Links between APIs and Security Vulnerabilities
Chapter 7: API Trustworthiness: An Ontological Approach for Software Library Adoption
user survey of 53 developers
integrated source code model (SEON) with SBSON
impact of breaking changes across project boundaries
SV-AF: integration of SEVONT, SEON, and SBSON
ontology alignment
impact analysis of security vulnerabilities across project boundaries
extended the MARKOS license ontology
introduced new trustworthiness measures
OntTAM: integration of all our ontologies
holistic trustworthiness analysis of reused libraries

Figure 1.1: An overview of the thesis content

Chapter 2

2 Motivation

Although the reuse of third-party libraries provides developers with gains in productivity by not having to re-implement already existing functionality, new technical and organizational challenges arise from this form of code reuse [11]. Some of these challenges identified in existing work include, but are not limited to, the following:

- choosing the most relevant library among several alternatives [12], [13],
- how to use features provided by these libraries [13], [14],
- cost of migrating to a new library [15], [16],
- maintenance costs due to breaking changes [17]–[19],
- impact of security vulnerabilities and bugs [20], [21],
- incompatible software licenses [5], [22], and
- unmaintained or outdated libraries [20], [23].

To address these challenges, existing approaches analyze the knowledge within software related repositories such as dependency repositories (e.g., Maven Central, npm), source code repositories (e.g., GitHub[4]), vulnerability databases (e.g., NVD[5]), and Q&A forums (e.g., StackOverflow). However, as mentioned in the introduction, most of these approaches treat these repositories as information silos and lack the ability to integrate their analysis results with existing knowledge, to make their analysis shareable and reusable for future analysis (both by humans and machines).

The motivation of this research is to establish a unified machine and human-readable representation that captures build and dependency information as well as knowledge from other software artifacts, to allow for a seamless knowledge integration across resource boundaries. This modeling approach will enable us, to transform the traditional information silos in which

[4] https://github.com/
[5] https://nvd.nist.gov/

these knowledge resources have remained into information hubs. Some of the key characteristics of such information hubs include the provision for standardized knowledge representation, cross-artifact analysis, and the reuse and sharing of analysis result across artifact and project boundaries.

The following motivating scenarios illustrate how such an integrated knowledge modeling approach not only allows for knowledge integration but can also provide the basis for novel types of software analytics.

Scenario #1: Bi-directional dependency analysis. Current build tools provide support for automatic dependency management; a project needs only to specify the third-party libraries it directly depends on, and the build tool automatically includes any required transitive dependent components. However, as shown in Figure 2.1(a), such dependency analysis only supports project-specific dependency trees based on unidirectional dependencies. While unidirectional dependency models work well for managing build dependencies, they are limited in their expressiveness and therefore restrict further reasoning upon the modeled information. For example, Maven's native support for impact analysis allows a developer to identify all the components used by his project, which is illustrated in Fig. 2.1(a). In this example, a component C depends on components D and E. However, given Maven's existing dependency model it would be impossible for an API producer to identify which projects depend (either directly or indirectly) on his API. A user study we conducted with open source developers indicated that library producers make better choices regarding breaking changes when they know the popularity of a library and how client projects use its APIs. Details of this user study can be found in section 5.3.

Using SW and its supporting technology stack, we can mine and model the dependencies of several projects to create a "global" bi-directional dependency graph (Figure 2.1(b)). As the figure illustrates, based on this enrich knowledge model, library producers can now easily identify all components which depend (directly or indirectly) on their libraries. For example, the developers of component C can identify components A, F, and G as clients which will be potentially impacted by any changes to C.

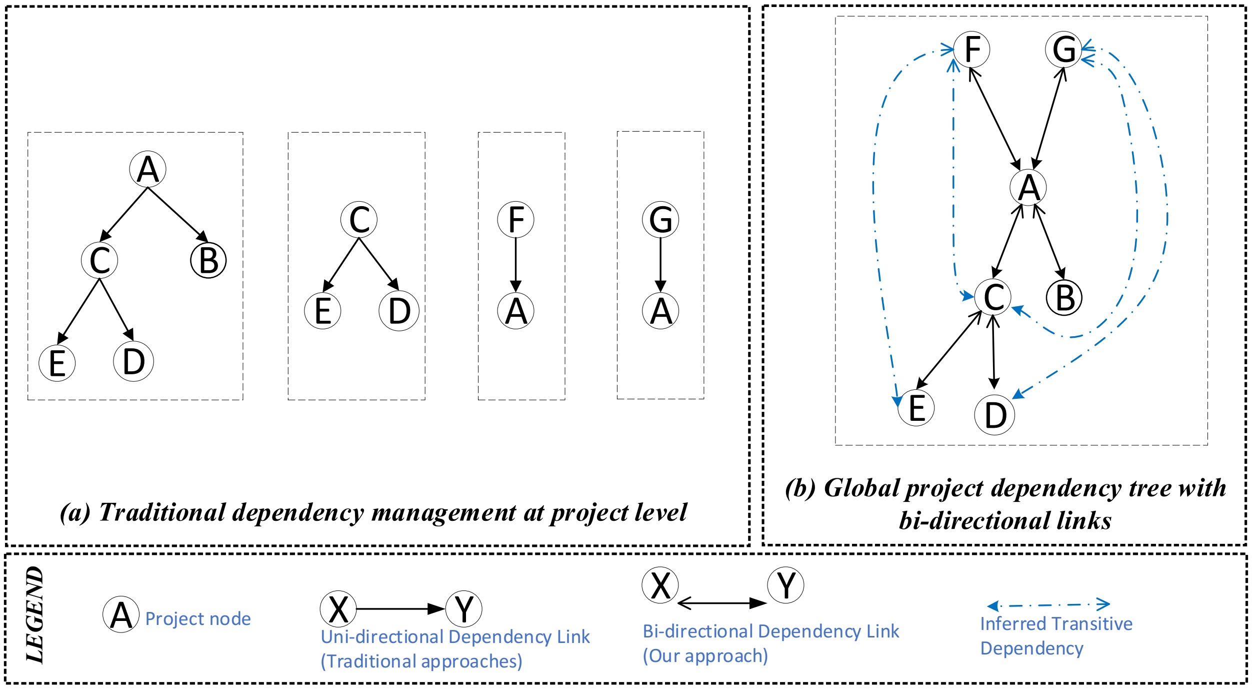

(a) Traditional dependency management at project level
(b) Global project dependency tree with bi-directional links
LEGEND
Project node
Uni-directional Dependency Link (Traditional approaches)
Bi-directional Dependency Link (Our approach)
Inferred Transitive Dependency

Figure 2.1: Overview of motivating scenario #1

Scenario #2: Supporting cross-artifact analysis. Many software analysis tasks extend beyond the source code and involve other software artifacts. For example, analysis tasks such as license violation detection and vulnerability impact analysis integrate knowledge from source code, license files, and vulnerability databases. While existing approaches and tools aim to support such types of analysis using project dependencies (e.g., VersionEye[6], SourceClear[7], OWASP-DC[8]). These approaches base their analysis on the existing knowledge representation (e.g., uni-directional dependencies for build management tools) of each individual knowledge sources, therefore treating them as information silos, which limits the analysis they can perform on the available knowledge.

In contrast, our approach takes advantage of SW and its supporting technologies to establish traceability through a global project knowledge graph. This graph integrates concept and facts from other software knowledge models, while supporting the inference of new knowledge and making analysis results an integrated part of the knowledge model. For example, in Figure 2.2, a

[6] https://www.versioneye.com/
[7] https://www.sourceclear.com/
[8] https://jeremylong.github.io/DependencyCheck/

traceability link is established between the two project E instances in the vulnerability and dependency models. We can now infer that projects C, A, F, and G are potentially vulnerable due to their transitive dependence on project E. Further, project A can be identified to introduce a license violation – with project A being transitively depends on project D which has a conflicting license.

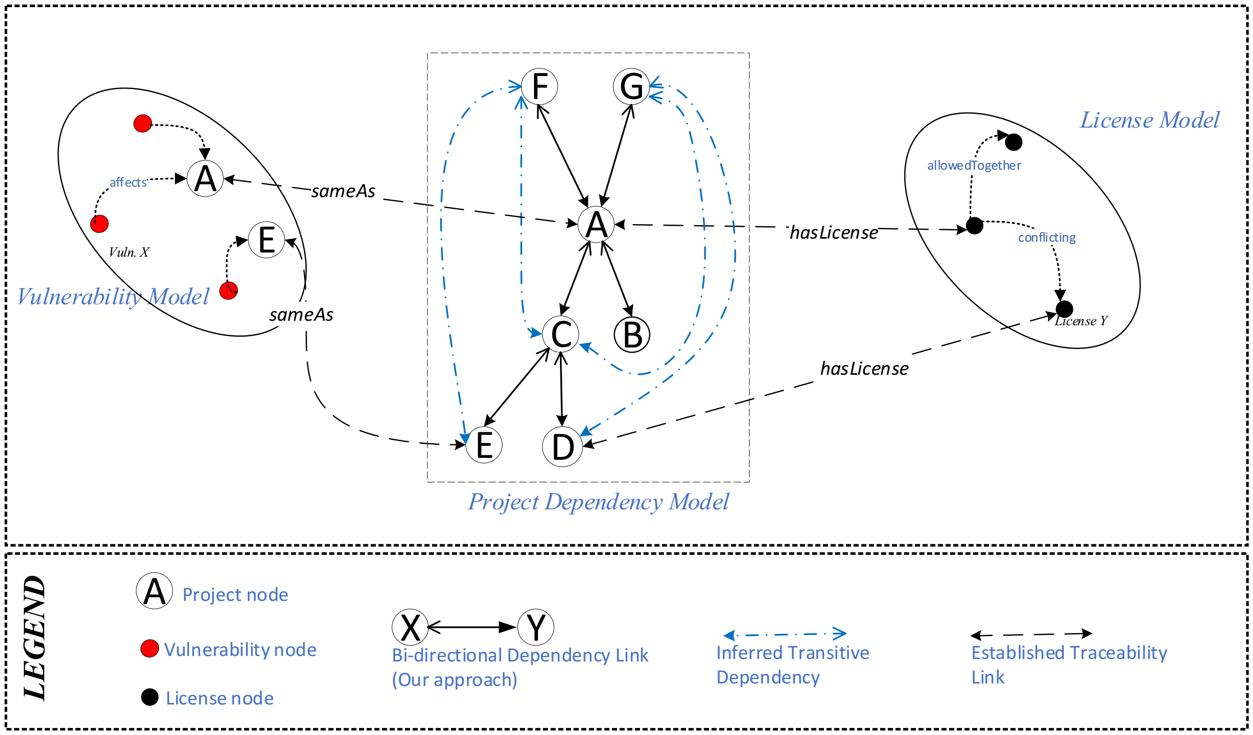


Figure 2.2: Overview of motivating scenario #2 - Integrating build information and knowledge from heterogeneous software repositories

As illustrated by the two scenarios, taking advantage of SW provides us not only with the ability to integrate distributed knowledge resources but also supports the Open World Assumption[9] (OWA), which must hold when modeling and analyzing these resources to be able to deal safely with incomplete data. That is, the lack of information cannot be used to infer further knowledge, which is in contrast to most existing source code analysis approaches which are based on the closed world assumption [24]. For example, in Figure 2.2, we do not have any established traceability link between project F's instance in the dependency model and the

[9] https://en.wikipedia.org/wiki/Open-world_assumption

vulnerability model. This does not mean project F has no security vulnerabilities; we cannot infer that fact at the moment. Also, using the Semantic Web, we can not only safely deal with incomplete data, but also support incremental knowledge population and take advantage of inference services provided by SW [25], [26]

One of the objectives of our approach is to provide links and inferences between existing knowledge resources and seamlessly integrate analysis results, to allow other analysis task to reuse already available results. For example, results of a vulnerability analysis can become an integrated (explicit) part of project related knowledge; other services can now reuse such results as part of their analysis without re-executing the initial vulnerability analysis.

Before introducing in detail our contributions, we will discuss some background relevant to our work.

Chapter 3

3 Background and Related Work

The work presented in this research combines different areas of Software Engineering (SE), including build systems and dependency management, MSR, and knowledge modeling. In this chapter, we provide a brief overview of core techniques, terminologies, and existing efforts in these fields that are related to our research. If you are already familiar with these concepts, you can safely move on to the next chapter as cross-references are provided throughout the thesis, whenever specific background information is required.

3.1 The Semantic Web in a Nutshell

Berners-Lee et al. define the Semantic Web as "an extension of the Web, in which information is given well-defined meaning, enabling computers and people to work in cooperation" [27]. In a Semantic Web, data can be processed by computers as well as by humans, including inferring new relationships among pieces of data. For machines to understand and reason about knowledge, this knowledge needs to be represented in a well-defined, machine-readable language.

The Semantic Web makes use of a set of technologies, frameworks, and notations defined by the World Wide Web Consortium (W3C) to be able to provide such formal description of concepts, terms, and relationships within a given knowledge domain. The Semantic Web is built around the central concept known as Ontology. Ontologies provide a formal and explicit way to specify concepts and relationships in a domain of discourse. They are a standardized platform for sharing vocabulary and knowledge to automate access and ease of use. Classes (and subclasses) are used to model concepts in ontologies, with properties modeling the attributes of such concepts.

Figure 3.1 provides an overview of the complete Semantic Web architecture and technology stack. The first (bottom) layer, URI, and Unicode are essential features of the existing WWW.

Uniform Resource Identifier (URI) allow to uniquely identify resources (e.g., documents) with Uniform Resource Locator (URL) being a subset of URI. The usage of URIs is essential for a distributed internet system as it provides understandable identification of all resources. XML is a general-purpose markup language for documents containing structured information and provides with its XML namespace and XML schema definitions a common syntax used by the Semantic Web.

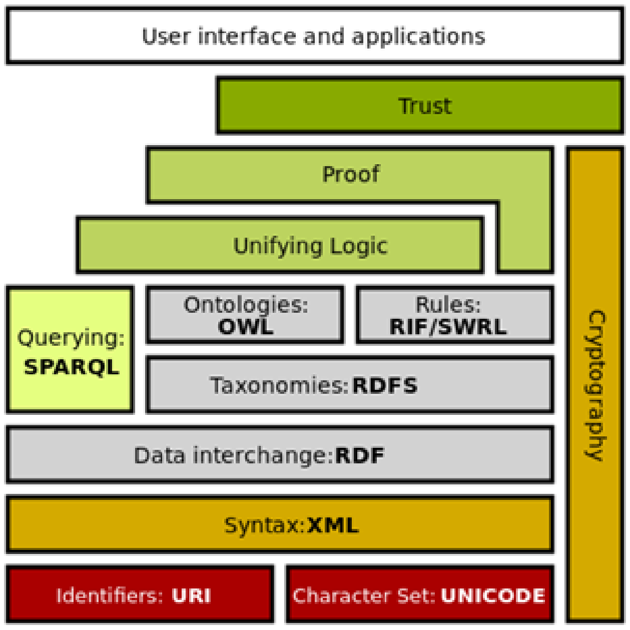


Figure 3.1: Semantic web architecture in layers

The Semantic Web uses the Resource Description Framework (RDF) as its underlying data model to formalize the meta-data as subject-predicate-object triples, which are stored in triple-stores. Triple-stores are Database Management Systems (DBMS) which model RDF data as a graph where nodes (subject, object) are connected through edges (predicates). An RDF Schema (RDFS) is combined with the formal semantics within RDF to allow for a standardized description of taxonomies and other ontological constructs. RDFS defines a simple modeling language on top of RDF which includes classes, “is-a relationships” between classes and between properties, and domain/range restrictions for properties. RDFS can be used to describe taxonomies of classes and properties and use them to create lightweight ontologies.

More detailed ontologies can be created through the use of the Web Ontology Language (OWL). OWL is derived from description logics (such as conjunction and disjunction, existentially and universally quantified variables), is syntactically embedded into RDF, so like RDFS, it provides additional standardized vocabulary. OWL comes in three forms - OWL Lite for taxonomies and simple constraints, OWL DL for full description logic support, and OWL Full for maximum expressiveness and syntactic freedom of RDF. RDFS and OWL have a set of defined semantics used for reasoning within ontologies and knowledge bases described using these languages. Standardized rule languages (e.g., Rule Interchange Format (RIF) and Semantic Web Rule Language (SWRL)) provide rules beyond the constructs available in RDFS and OWL. With these rule and logic constructs, a reasoning module can make logical inferences and derive knowledge that was previously only implicit in the data. Using OWL for the Semantic Web implies that an application could invoke such a reasoning module and acquire inferred knowledge rather than only retrieve data [28]. For querying RDF data as well as RDFS and OWL ontologies with knowledge bases, a Simple Protocol and RDF Query Language (SPARQL) is available. Since both RDFS and OWL are built on RDF, SPARQL can be used for querying ontologies and knowledge bases directly as well. SPARQL is a query language for RDF which attempts to match patterns in the RDF graph to find solutions [29].

In the Semantic Stack, it is expected that all semantics and rules will be executed at the layers below Proof and the result will be used to prove deductions. Formal proof together with trusted inputs for the proof will mean that the results can be trusted. For reliable inputs, cryptography means are to be used, such as digital signatures for verification of the origin of the sources. On top of this technology stack, is end-user interfaces and application that take advantage of the Semantic Web infrastructure.

Linked data [30], [31] is a by-product of the Semantic Web. It was introduced to ease data sharing and integration in distributed environments and be superior to XML-based approaches [32], [33]. Linked data is mainly about publishing structured data in RDF using URIs rather than focusing on the ontological level or inferencing. Linked Data best practices have led to the extension of the Web with a global data space which allows for connecting data from diverse domains, such as online communities, statistical and scientific data. Linked data enables both humans and machines to interpret data for mining, searching, and analysis purposes. Each entity in the domain of discourse must have a unique identifier (UID) in the form of a URI (Uniform Resource Identifier). Linked data mandates that URLs are de-referenceable to make information inter-linkable and online. That is, clients (i.e., humans and machines) must be able to fetch resource-related data via its URL (with the http:// prefix). Using an HTTP header, a client specifies the desired output format: HTML or RDF/XML.

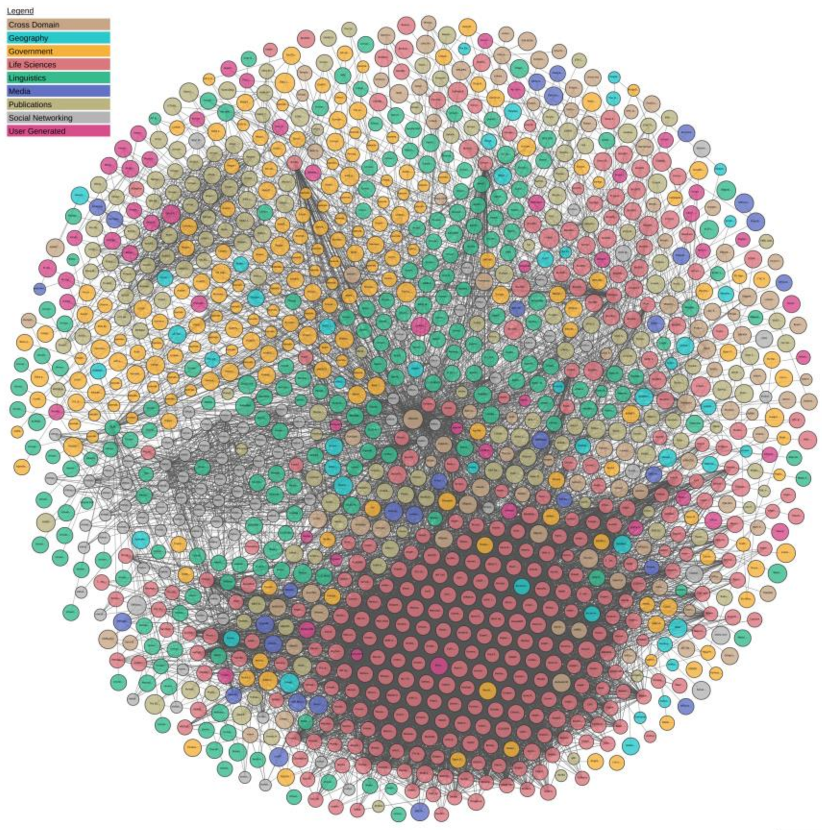


Figure 3.2: State of the LOD cloud[10]

3.2 Ontologies in Software Engineering

Representing software in terms of knowledge rather than data, ontologies provide a better support for representing the semantics of software [27] compared to relational databases where sharing and reuse of schemata are not natively supported. Semantic Web meta-models are extensible, allowing the addition of new knowledge without affecting existing knowledge. Unlike relational databases, where extending the schema becomes a time-consuming operation, often affecting a complete database (e.g., changing a foreign key index type might require dropping and recreating several other dependent database indices). Among other benefits

[10] https://lod-cloud.net

identified by [34], are that the Semantic Web makes relations and their meaning explicit. Relational databases lack a consistent method for obtaining the semantics of a relation and therefore, a query can join any two table columns, if their datatypes match – there is no interpretation of the meaning of the relation performed. As a result, relational databases are not machine-interpretable, and the inference of knowledge (explicit or implicit) requires human interaction. Also, linking data is a vital property of the Semantic Web, with resources identified by their Uniform Resource Identifier (URI). These URIs, allow for consistent identification of the same resource across various knowledge resources. This contrasts with relational databases where resources are local and not universal, therefore restricting the ability of relational databases to establish resource links outside their local schema.

Given the current diversity in technologies and software development processes, produced software artifacts are often disconnected from each other. With the rate at which software project artifacts become available in (online) repositories, a common issue faced by programmers is the need to locate knowledge relevant to their specific development task. While the MSR community has made significant progress in analyzing individual repositories by introducing proprietary mining techniques, the MSR community has yet to address the issue of seamless integrating these knowledge resources [34]. Several approaches to establish taxonomies for software engineering through ontologies have been presented recently to describe domain knowledge of developers, source code, and other software artifacts. The common goal of these approaches is to foster reuse and support the automatic inference of new knowledge.

For example, in requirement engineering, ontologies have been used to support requirement management [35], traceability [36], and use case management [37]. In the software testing domain, KITSS [38] is a knowledge-based system that can assist in converting a semi-formal test case specification into an executable test script. For the software maintenance domain, Ankolekar et al. [39] provide an ontology to model software, developers, and bugs. The authors developed a prototype Semantic Web based system, Dhruv, which provides an enhanced semantic interface to bug resolution messages and recommends related software objects and artifacts for the OSS community. Ontologies have also been used to describe the functionality of components using a knowledge representation formalism that allows more convenient and powerful querying. For example, the KOntoR [40] system stores semantic descriptions of components in a knowledge base and supports the semantic querying of this knowledge. In [41], Jin et al. discuss an ontological approach of service sharing among program comprehension tools. Hyland-Wood et al. [42] proposed an OWL ontology of software engineering concepts, including classes, tests, metrics, and requirements. Bertoa et al. [43] focused on software measurement. Witte et al. [44] used text mining and static code analysis to map documentation to source code in RDF for software maintenance purposes. Yu et al. [45] also represented static source code information using an OWL ontology and used SWRL rules to infer common bugs in the source code.

Several researchers have described software evolution artifacts extracted from existing software repositories as OWL ontologies. Their approaches integrate different artifact sources to facilitate everyday repository mining activities. Kiefer et al. presented EvoOnt [46], an integration of a code ontology model, a bug ontology model, and a version ontology model used to detect bad code smells and extract data for visualizing changes in code over time. Iqbal et al. presented their Linked Data Driven Software Development (LD2SD) methodology [47] to provide RDF-based access to JIRA bug trackers, Subversion, developer blogs, and project mailing lists. Wursch et al. presented SEON [34], a family of ontologies that describe many different facets of a software's lifecycle. SEON is unique in that it comprises of multiple abstraction layers.

Like SEON, our approach organizes ontologies in consecutive layers of abstractions with clear representational purpose. We also extend existing source code ontologies and introduce a taxonomy for describing dependency management semantics. Due to the uniform RDF format used by these approaches, we can envision interesting interactions among our semantics-aware analysis and ontologies introduced by others. Such extensions could lead to an entirely new family of software analysis services or at least simplify the implementation of existing ones.

3.3 Mining Software Repositories (MSR)

A software repository commonly refers to a persistent storage location where artifacts related to software projects and their development lifecycle are stored. Such repositories are used to record daily interactions between the stakeholders, as well as the evolutionary changes to various software artifacts. Mining Software Repositories (MSR) is a field of software engineering research which aims to analyze and provide additional insights in the data stored in these software repositories. The main goal of MSR is to make use of historical data in these repositories and transform it to become actionable data that can support various decision-making processes during software development [48]–[51]. Research has shown the importance of MSR in software development decision making in several areas including bug identification and prediction [52], [53], understanding team dynamics [54], [55], improving user experience [56], and code reuse [57]. Table 3.1 provides a general overview of SW repositories and how the MSR community uses these repositories to mine them to derive actionable facts.

Table 3.1: Examples of Software Repositories

Repository	Description	Example MSR applications
Source code repositories	These repositories archive the source code for a large number of projects. Sourceforge[11] and GitHub are examples of such large code repositories.	Source code differencing and analysis [58] Factors for successful software reuse [59] Inter-project collaboration [60], [61]
Bug/Issue repositories	These repositories track the resolution history of bug reports or feature requests that are reported by users and developers of large software projects. Bugzilla[12] and Jira[13] are examples of bug repositories	Relationship between bugs/features [62] Automated bug assignment [63]
Archived communications	These repositories track discussions about various aspects of a software project throughout its lifetime. Mailing lists, emails, IRC chats, and instant messages are examples of archived communications about a project	Why developers join and leave a project [54] Immigration in open source systems [55]
Version Control	These repositories record the development history of a project. They track all the changes to the source code along with meta-data about each change, e.g., the name of the developer who performed the change, the time the change was performed and a short message describing the change. Source control repositories are the most commonly available and used repository in software projects. GitHub and BitBucket[14] are examples of version control repositories which are used in practice	Change prediction [64]–[66] Call-usage patterns [67], [68] Change patterns [69] Characteristics of different types of changes [70] Incomplete refactoring [71] Code search [72] Clone detection [73]

[11] https://sourceforge.net/
[12] https://www.bugzilla.org/
[13] https://www.atlassian.com/software/jira
[14] https://bitbucket.org/

Programming Question and Answer (Q&A) Repositories	These repositories allow developers to get help with their code by posting questions and answering each other's questions. They keep track of all questions and answers, as well as meta-data about users and votes. StackOverflow and CodeProject [15] are examples of programming Q&A repositories.	Predicting how long a question will remain unsolved [74] Finding a good code example [75] Study on personality traits of Q&A users [76] Developer interactions (Wang, Lo, and Jiang 2013; Barua, Thomas, and Hassan 2014)

3.4 Build Systems and Dependency Management

Build systems transform the source code of a software system into deliverables. There are several build technologies available for developers, and they adopt different design paradigms [79]. The four most common build paradigms as defined by [80] are:

i. *Low-level technologies*. These require explicitly defined dependencies between each input and output file (e.g., Make[16], Ant[17]).

ii. *Abstraction-based technologies*. These use high-level abstractions to automatically generate low-level build specifications; this addresses the portability flaw faced by platform-specific low-level technologies (e.g., CMake[18]).

iii. *Framework-driven technologies*. These favor build conventions over configuration. Such build technologies assume that if projects abide by these conventions, then build behavior can be inferred automatically (e.g., Maven).

iv. *Dependency management technologies*. These support the three above paradigms by automatically managing external API dependencies. This offers the advantage of users no longer needing to manually install all required versions of libraries (e.g., Ivy[19]).

Despite the different design paradigms, all build systems capture the build process – a process by which software can be incrementally rebuilt, allowing developers to focus on making source code changes rather than having to worry about managing a project's build dependencies. Build processes can be split into four steps [79]. First, a set of user or environment features is

[15] https://www.codeproject.com/
[16] https://www.gnu.org/software/make/
[17] https://ant.apache.org/
[18] https://cmake.org/
[19] https://ant.apache.org/ivy/

selected during the configuration step. Next, the construction step executes the compiler; code transformation commands that produce deliverables are executed in an order such that dependencies among them are not violated. The certification step follows automatically executing tests to ensure that the produced deliverables have not regressed. Finally, the packaging step bundles certified deliverables together with required libraries, documentation, and data files. These steps and information on all needed dependencies are defined in one or multiple build files and stored in specialized build repositories to facilitate reuse and sharing. The most popularly used build repositories for open source projects include Maven Central, npm, PyPi[20], and RubyGems.

Since transforming source code into a usable artifact is the main goal of build systems, source code evolution may act as a catalyst to the evolution and maintenance of build systems. Adams et al. [81], [82] and Godfrey et al. [83] studied the static evolution of the Linux kernel build system, which is implemented using make. They found that the Linux kernel build system is growing exponentially in terms of the Build Lines of Code (BLOC). Furthermore, the build and source code appear to grow together or shrink together, suggesting that source code and build system co-evolve. McIntosh et al. [84] further show that this co-evolution imposes an overhead on the development process. They examine how frequently source code changes require build changes and the proportion of developers responsible for build maintenance. Their results indicate that build changes induce more churn on the build system than source code changes induce on the source code. Furthermore, build maintenance yields up to a 27% overhead on source code development and a 44% overhead on test development, with up to 79% of source code developers and 89% of test code developers significantly being impacted by build maintenance.

Although source code and build systems co-evolve together, due to the complex nature of build systems, it is still difficult to identify when source code changes require accompanying build changes (build co-changes). McIntosh et al. [6] mined random forest classifiers from historical data using language-agnostic and language-specific code change characteristics to explain when code-accompanying build changes are necessary. Their results suggest that most C++ build changes and at least the code-related Java build changes can indeed be predicted using

[20] https://pypi.org/

characteristics of corresponding changes to source and test code. Xia et al. [7] also propose an approach which predicts when such build co-changes are necessary. Their approach, however, also considers the "cold-start" problem for new projects when there exists only a limited number of changes which can be mined. They use training data from other projects to predict build co-changes in a new project (transfer learning).

Beyond the study of build evolution and maintenance, several researchers have demonstrated how mining build repositories can benefit a variety of software tasks such as: identifying license compliance inconsistencies; identifying build cloning; and automatic library recommendation and migration. In [5], the authors proposes an approach to construct and analyze the system calls that occur at build-time to study license violations. A concrete build dependency graph is created by tracing OS calls made by the build tools during execution. This makes it easy to identify which source files are being used, which external components are called and how the code and components are combined. Through labeling each source file node in the graph with its corresponding license, license violations can be identified.

McIntosh et al. [8] study how much cloning occurs in build systems and whether these clones are affected by technology choices. They gauge cloning rates in build systems by collecting and analyzing a benchmark comprising 3,872 build systems. Their results reveal that build systems tend to have higher cloning rates than other software artifacts, and recent build technologies are often more prone to cloning, especially the configuration details like API dependencies, compared to older technologies.

Another interesting application of build system knowledge is in automatic library recommendation and migration. With the growing rate at which third-party libraries are reused, dependency management has become a feature adopted in most current build systems. However, they lack support for library recommendations that would guide developers in selecting which library (and its version) to be used. Mileva et al. [2] propose an approach which uses historic trends of library usages within the Maven Central repository to recommend the most commonly used library as the most suitable to adopt. Teyton et al. [16] mine the Maven Central repository to build migration graphs for different categories of libraries. With these graphs, one can quickly identify which libraries are the best candidates to migrate to. When recommending library adoptions and migrations, backward compatibility becomes a very desirable trait. One way to inform library users of the level of compatibility of a library is through its version number – a *major.minor.patch* versioning scheme suggested by semantic versioning[21]. Raemaekers et al. [85] analyze a dataset of 150,000 Maven libraries to determine if these versioning numbering rules are adhered to. Their results show that there has been only a marginal increase in the adoption of this scheme over time. The impact of not adopting these versioning rules is highlighted by their results; a third of all releases introduce at least one breaking change. The authors concluded that version numbers currently do not provide developers with enough information on the stability of library interfaces.

As discussed earlier, build systems are an essential part of software systems. Among others, they control variability and manage configurations, deciding which files and features to include in the compiled product. Many tools have been introduced to extract this configuration knowledge to analyze and maintain highly configurable systems. However, with the increasing number of configuration options and complexity of build systems, build scripts become also more complex; making it harder to understand, analyze and maintain a build system. In this section, we discuss related research approaches which focus on the extraction and analysis of configuration in build systems in terms of file presence conditions and conditional parameters.

Most of the reviewed analysis approaches are dynamic; they derive their analysis data from the execution of the build scripts. For example, van der Burg et al. [5] dynamically detect which files are included in a build to check license compatibility, Metamorphosis [86] dynamically analyzes build system to migrate them, and MkFault [87] combines runtime information with some structural analysis to localize build faults. However, such dynamic approaches can only analyze file presence conditions of one configuration at a time. While a dynamic analysis which involves the execution of all possible configurations would yield accurate variability information, such an approach does not scale.

Alternatively, researchers have also applied different types of static analysis on build files. SYMake [88] uses symbolic execution to conservatively analyze all possible executions of a Makefile. This approach produces a symbolic dependency graph, which represents all possible build rules and dependencies among targets and prerequisites, as well as recipe commands. This approach can be used to detect different types of errors in Makefiles and help to build refactoring tools. Dietrich et al. [89] sample a subset of configurations, trying to activate each configuration

[21] http://semver.org/

option once. Their approach is simple due to its sampling nature, but incomplete; it cannot recover complex conditions with several disjunctions and negations. Using a different strategy, both Berger et al. [90] and Nadi and Holt [91] have tried to statically approximate file presence conditions by detecting specific patterns in build scripts used in Linux's Kbuild infrastructure. These approaches achieve relatively high precision for the Linux kernel but are unable to cope with build files (or parts thereof) that do not follow these patterns. [92] work builds on SYMake; they are specifically interested in extracting variability information in terms of file presence conditions and conditional parameters. In their approach, the authors' use symbolic execution which does not rely on sampling or specific patterns.

3.5 Chapter Summary

In this chapter, we provided background information for core technologies and concepts used in our research and reviewed the existing research in these areas. We will frequently refer to this chapter in subsequent chapters.

In the next chapter, we discuss in more detail the knowledge engineering process we applied to create our unified ontological representation for build and dependency management semantics. This unified representation provides us with the foundation for our seamless integration of dependency knowledge into existing SE development tasks.

Chapter 4

4 A Unified Ontology-based Modeling Approach for Software Build and Dependency Repositories

As mentioned before, the overall goal of this thesis is to introduce a novel semantic software build and dependency knowledge model, which allows for knowledge integration with other software artifacts and supports novel knowledge-driven dependency analysis services. More specifically, we introduce an ontology for the domain of build and dependency management systems which supports reasoning and inferencing of new knowledge. In addition, the expressiveness and flexibility of our ontology allows for knowledge reuse and sharing, and a seamless integration of build dependency knowledge into existing SE development tasks.

In this chapter, we explain the knowledge engineering methodology which we applied to the construction of our unified knowledge model and the design decisions we made to address some of the open research challenges identified in our research motivation (Chapter 2).

4.1 Introduction

As discussed in Section 3.4, build systems adopt different design paradigms, structures, and syntax to transform source code using user-defined build processes. Common to these build systems is a set of core semantics. One of the main objectives of this research is to abstract and formally model the domain of build and dependency management in a technology-independent representation. In this chapter we introduce a semi-automated approach for the development of a software build dependency domain ontology, which is based on the discovery, reuse, and integration of knowledge from existing build repositories. More specifically, our methodology takes advantage of SW technologies to provide a standardized and unified representation of build dependency semantics. The proposed model adheres to the following design criteria proposed by [93], [94].

- *Unambiguous Semantics*. The primary motivation for using ontologies over other modeling approaches is to enrich information with semantics. The absence of clear semantics may lead otherwise to diverging interpretations of intended meaning. Formalism, through defining concepts with logical axioms, is the means to this end. To the best of our knowledge, there exists currently no semantic vocabulary for describing build and dependency management systems; the presented knowledge model in this thesis is the first formal semantic vocabulary developed for the build and dependency management domain.
- *Extendibility*. Our model design considers easy extensibility of our ontologies; the addition of new concepts does not require the revision of the existing definitions.
- *Reasoning and Inferencing*. Our ontology design provides support for basic semantic reasoning and inferencing (e.g., RDFS++ reasoning). The model supports different types of reasoning within and across the ontology in order to support a seamless integration of knowledge resources at different abstraction levels. Instead of building our model based on general inferencing, we use lightweight reasoning such as Open World Assumption, classification, transitivity and consistency. These, compared to general inferencing sustain the scalability and tractability of our model [93]. Details of the reasoning capabilities supported by our model can be found in Chapters 5 to 7.

4.2 Software Build System ONtology (SBSON): Knowledge Modeling and Engineering

Different knowledge engineering methodologies have been discussed in the literature (e.g., Noy et al. [95], Van der Vet et al. [96], and Uschold et al. [97]. Noy et al. [95], in their knowledge-engineering approach for ontology development, proposed the following seven core steps: (1) determining the domain and scope of the ontology, (2) considering the reuse of existing ontologies, (3) enumerating essential terms in the ontology, (4) defining the classes and class hierarchy, (5) defining the properties of class-slots, (6) defining the facets of the slots, and (7) creating instances. Van der Vet et al. [96] proposed a bottom-up approach for building ontologies. Their approach depends on atomism, that is, the idea that objects are composed of indivisible units called "atoms." They use part-whole relations to group basic concepts into "superconcepts".

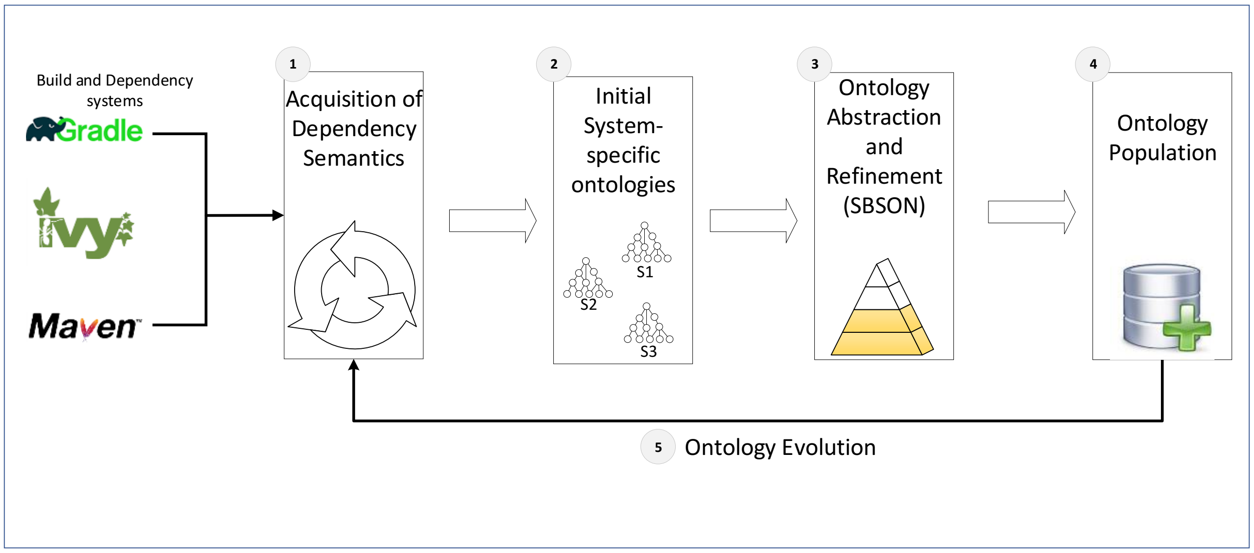

Build and Dependency systems
Gradle
Ivy
Maven
1
Acquisition of Dependency Semantics
2
Initial System-specific ontologies
S1
S2
S3
3
Ontology Abstraction and Refinement (SBSON)
4
Ontology Population
5
Ontology Evolution

Figure 4.1: An overview of our knowledge modeling methodology.

Our methodology which consist of five major steps (Figure 4.1), is based on the methodology introduced by Noy et al. [95] and a bottom-up knowledge modeling approach similar to the one used by Van der Vet et al. [96]. We first perform a manual review of the documentation from selected build and dependency management systems and their repository structure to identify and extract concepts and properties used by the individual build dependency management systems. Next, in Step 2, we manually inspect these extracted concepts and properties for each build system to derive an initial version of the corresponding system-specific ontologies. After creating these system-specific ontologies, Step 3 uses a bottom-up approach to identify and extract shared concepts and attributes from these system-specific ontologies into different layers of abstraction (upper ontologies). We then further refined and enriched the initial design of these ontologies, by adding additional relations and properties, to have a model semantics which is rich enough to allow for the inference of knowledge using basic SW reasoning (RDFS++). We then populate, in Step 4, these newly created knowledge model with facts from projects published in open source build repositories. Finally, during the last step of our methodology, we evolve our ontologies with new build and dependency management systems and concepts as they become available.

The outcome of this modeling process is a comprehensive ontology that captures the domain of build and dependency knowledge. The final layered model is based on a meta-meta model approach (e.g., Object Management Group (OMG)[22]), where the top layer captures the core elements, which are extended and refined throughout the abstraction hierarchy. Figure 4.2 presents an overview of the different ontology abstraction layers in SBSON. For a complete description of these ontologies, we refer the reader to [98].

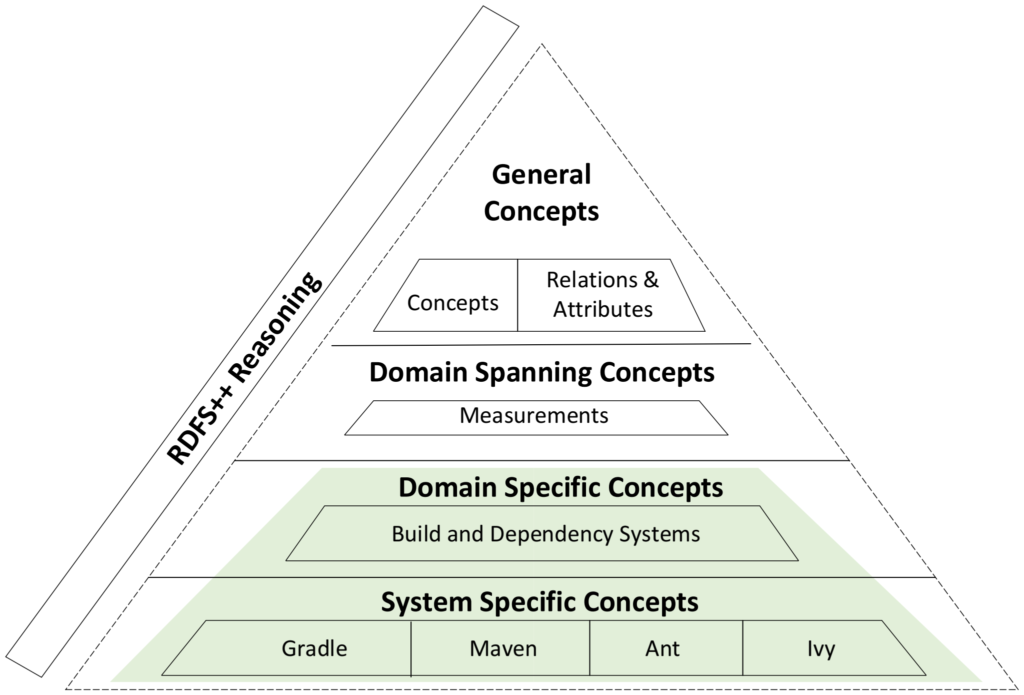


Figure 4.2: An overview of the different ontology abstraction layers in SBSON.

Within our knowledge hierarchy, the *General Concepts* layer captures the omnipresent core concepts related to software evolution. The *Domain-Spanning Concepts* layer builds upon the

[22] http://www.omg.org/

General Concepts layer and captures concepts that span across several subdomains in our model (e.g., vulnerability databases, version control systems, and source code). Concepts within this layer are introduced in later chapters when other SE knowledge sources are integrated with SBSON. The concepts at the *Domain-Specific* layer are common to resources in a domain, such as software build and dependency concept. Finally, the *System-Specific* layer's concepts represent knowledge that is specific to a given data source or system and not commonly shared across the domain. In Chapters 5 to 8, we discuss in detail how SBSON can be integrated with other SE knowledge sources such as source code, version control systems, and vulnerability databases. In what follows, we describe in detail the five major knowledge modeling steps which we applied in our approach.

4.2.1 Step 1: Acquisition of Dependency Semantics

Most build systems are based on a formalized syntax and structure, which can be further customized through configurations. With this in mind, we conducted a survey of three (3) popular Java build management systems from different vendors which make use of the same build repository, Maven Central, to store and resolve project dependencies. We are especially interested in finding how different dependency management features are implemented in each studied system. An overview of these three systems is provided in Table 4.1 and general statistics of the Maven Central repository is provided in Table 4.2. It should be noted that, although we only studied systems which utilize the Maven Central repository, our knowledge modelling approach provides the flexibility to extend and evolve our ontologies with different build systems and repositories. Details of our ontology evolution step can be found in Section 4.2.5.

Table 4.1: Overview of the 3 studied build and dependency management systems

ID	**Name**	**Vendor**	**Default repository**	**Dependency management features**				
				Transitivity	***Filtering***	***Version Ranges***	***Scope***	***Default Resolution***
S1	Ivy (with Ant)	Apache	Maven Central	YES	YES	YES	NO	Latest version
S2	Gradle	Gradle		YES	YES	YES	YES	Latest version
S3	Maven	Apache		YES	YES	YES	YES	Nearest

Table 4.2: General statistics of the Maven Central repository

Repository	Identification Scheme	# Projects	# Releases	Snapshot Date
Maven Central	groupID-artifactId-version	279,853	3,687,307	2019-May-07

While the surveyed systems support dependency management features such as transitivity, dependency filters (exclusions), and version ranges, only Maven (S3) and Gradle (S2) support dependency scopes. Furthermore, because the Java Virtual Machine (JVM) is unable to differentiate between multiple API versions in a project's class-path, different conflict resolution techniques are used by the analyzed systems. For example, S1 and S2 choose (by default) during version conflict resolution always the latest version of a dependency, while S3 automatically selects the dependency version closest to the project's root definition (the version with the least transitive depth).

Among other features supported by these systems are multi-module projects and inheritance of dependency configuration from parent projects.

4.2.2 Step 2: Initial System-Specific Ontologies

Next, we manually identify and extract dependency related concepts and attribute definitions from the schemata and their documentation to create system-specific ontologies for each system. Figure 4.3 shows an overview of the three system-specific ontologies we extracted.

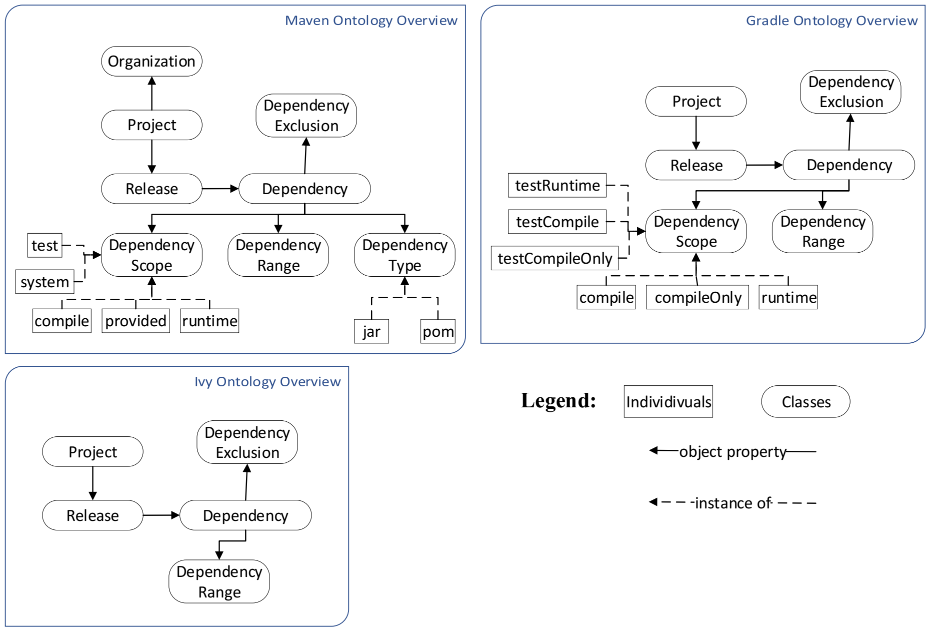


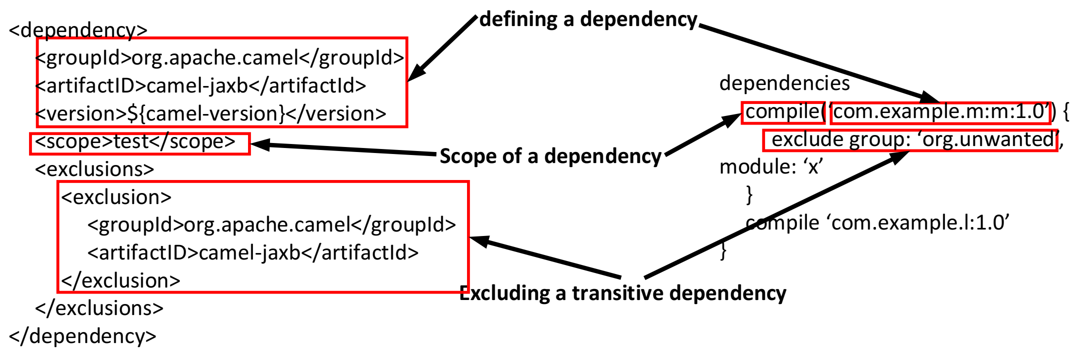
Figure 4.3: Overview of individual system-specific ontologies for the analyzed systems.

A main challenge we had to deal with during this analysis step was to identify and resolve the differences in syntax and structure of similar concepts and properties in the three systems. Table 4.3 and Figure 4.4 illustrate examples of such representation differences for version ranges and dependency definitions. As shown in Table 4.3, different symbols are used by Ivy and Maven when defining open and half-open intervals[23] for version ranges. For example, Ivy uses "]" to declare an open minimum version while Maven uses "(".

In the next section, we describe our knowledge modeling approach which we used to remove some of this ambiguity while extracting a domain (upper) ontology from the lower level system ontologies.

Table 4.3: Syntax differences for defining dependency version ranges

Version Range	**Ivy Syntax**	**Maven Syntax**	**Gradle Syntax**
Exact version	1.0	Same as Ivy	Same as Ivy
all versions greater than 1.0	]1.0,)	(1.0,)	Same as Maven
all versions greater or equal to 1.0	[1.0,)	Same as Ivy	Same as Ivy
all versions lower or equal to 2.0	(,2.0]	Same as Ivy	Same as Ivy
all versions lower than 2.0	(,2.0[	(,2.0)	Same as Maven
all versions greater than 1.0 and lower than 2.0	]1.0,2.0[	(1.0,2.0)	Same as Maven
all versions greater than 1.0 and lower or equal to 2.0	]1.0,2.0]	(1.0,2.0]	Same as Maven
all versions greater or equal to 1.0 and lower than 2.0	[1.0,2.0[	[1.0,2.0)	Same as Maven
all versions greater or equal to 1.0 and lower or equal to 2.0	[1.0,2.0]	Same as Ivy	Same as Ivy
all revisions starting with '1.0.' (e.g., 1.0.1, 1.0.a)	1.0.+	n/a	Same as Ivy



```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactID>camel-jaxb</artifactId>
  <version>${camel-version}</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.apache.camel</groupId>
      <artifactID>camel-jaxb</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

```
dependencies
compile('com.example.m:m:1.0') {
  exclude group: 'org.unwanted',
module: 'x'
  }
  compile 'com.example.l:1.0'
}
```

Figure 4.4: Example of syntax and structural differences between Maven (left) and Gradle (right) dependency definitions

[23] Open intervals do not include the declared minimum and maximum allowed versions of a dependency during dependency resolution; half-open intervals include only one of the declared range endpoints.

4.2.3 Step 3: Ontology Abstraction and Refinement

In this step of our knowledge modeling approach we use the extracted system-specific ontologies to abstract a software build-dependency domain ontology. This domain ontology promotes knowledge reuse through the identification of shared concepts and properties. It also allows for the linking of system-level ontologies with each other via the abstracted shared concepts and properties found in the domain ontology.

More specifically, this step identifies any concept or property that can be promoted from the System-specific to the Domain-specific layer of our knowledge model. For example, concepts related to transitive dependencies, dependency filtering, and version ranges can be promoted to the Domain-specific layer since they are shared among all three system-specific ontologies. The Domain-specific layer not only promotes such reuse of concepts across system level ontologies, but also improves traceability among system level ontologies by unifying the overall knowledge representation.

Although the identification of shared concepts can be considered somewhat as a straightforward task, reasoning capabilities are important requisites for inferring new knowledge and creating traceability links between domain and system-level ontologies.

In what follows, we describe in detail how we enrich our domain and system-specific ontologies with OWL reasoning capabilities (provided by the SW) and existing ontology design patterns. More specifically, we describe the modeling of (1) dependency links, (2) order of project releases, (3) version ranges, (4) dependency exclusions, and (5) transitive dependencies. It should be noted that in order to improve the readability, we use prefixes as substitutes to the fully qualified names of our ontologies. The ontology prefixes used in this chapter can be dereferenced using the URIs shown in Appendix A.

4.2.3.1 Modeling Dependency Links

Problem. As shown in Figure 4.5(a), a defined dependency between any two project releases can have additional associated characteristics such as the version range of the dependency and a list of excluded transitive dependencies. Since OWL does not natively support the definition of properties on top of other properties, modelling such dependency link characteristics becomes a challenge.

Solution. To address this challenge, we adopt the property reification design pattern[24]. In the following, we illustrate the use of the property reification pattern to model facts about the dependency relation between two project releases.

We introduce the <sbson:DependencyLink> concept to represent the dependency link between a source (with the <sbson:hasDependencySource> property) and a target (with the <sbson:hasDependencyTarget> property). The <sbson:DependencyLink> concept provides us with the flexibility of defining dependency-specific version ranges and exclusions as shown in Figure 4.5(b). This reification design approach provides us with an extensible and expressive modeling that can capture different characteristics of project dependency links. However, since a dependency is now modelled by the <sbson:DependencyLink> class, transitive reasoning on dependencies is no longer supported by default. We mitigate this problem by adding custom rules (explained in detail in section 4.2.3.5) which deduce transitive reasoning from the reification design pattern.

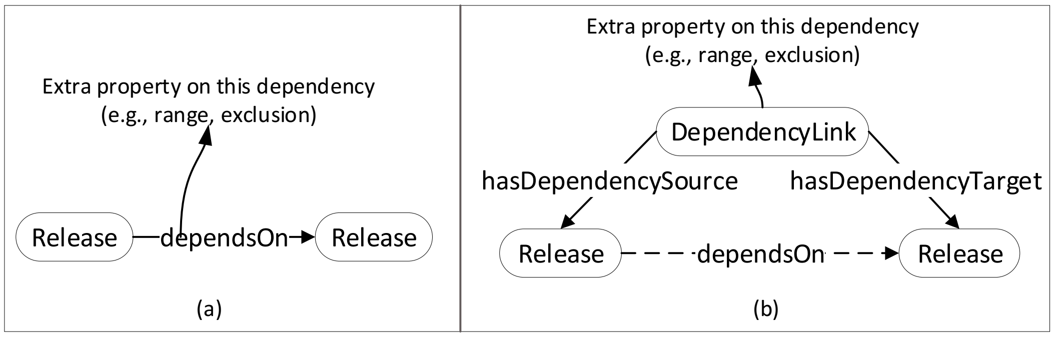


Figure 4.5: An illustration of (a) generic dependency between two releases, and (b) how property reification pattern is adopted in modeling dependency links

4.2.3.2 Modeling the Order of Project Releases

Problem. Software libraries use version numbers to uniquely identify their releases. These version numbers are assigned in an incremental order to define the order of releases and indicate backward compatibility (semantic versioning). In the context of dependency management, knowing the order of project releases is necessary for resolving dependencies related to version ranges. Unfortunately, the SW does not natively support ordered lists.

[24] https://www.w3.org/wiki/PropertyReificationVocabulary

Solution. We address this challenge by reusing the existing OrderedList Ontology[25] to model projects and the order of their releases. The OrderedList ontology, illustrated in Figure 4.6(a) consists of the <olo:OrderedList> and <olo:Slot> concepts. An ordered list is composed of a number of slots (using the <olo:slot> property). Items in an ordered list are associated to slots by the <olo:item> property and are accessed using the <olo:next> iterator property. Data properties such as <olo:length> and <olo:index> are used to represent the total number of slots in the list and the index of each slot respectively.

Figure 4.6(b) illustrates our extension of the OrderedList ontology which assigns one ordered list to each project. The multiple releases of a project are subsequently ordered by assigning them as items to slots of the project's ordered list. Figure 4.6(c) shows an example of a project with three ordered releases.

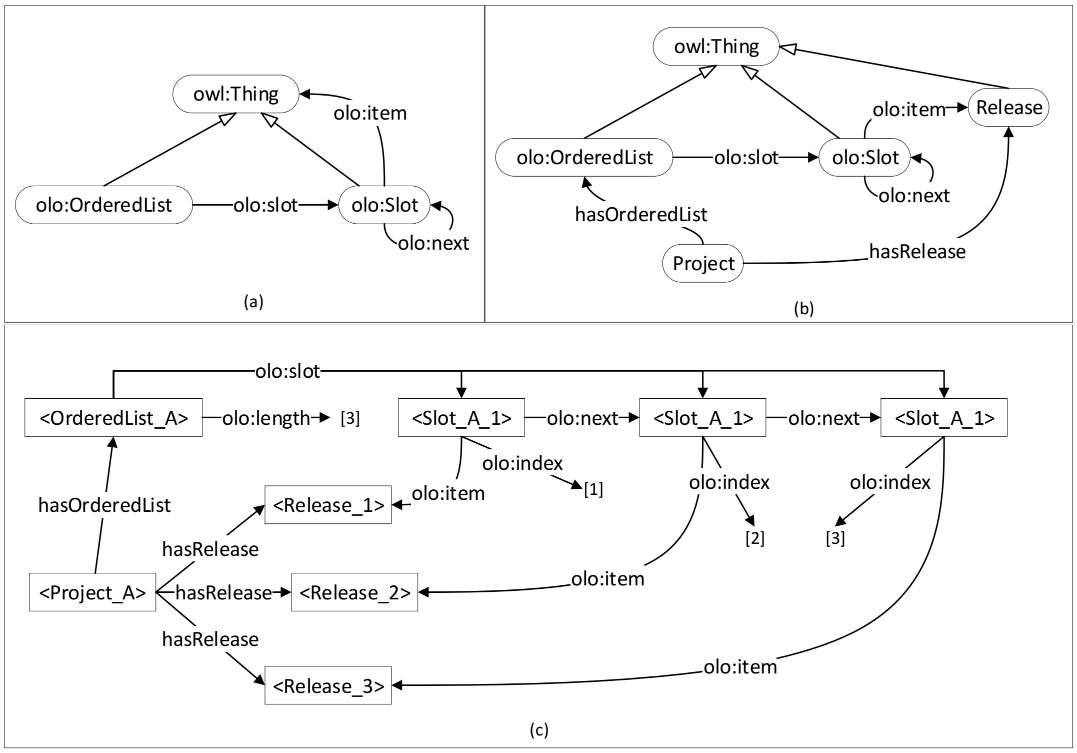


Figure 4.6: (a) The OrderedList Ontology, (b) how we model the order of project releases with the OrderedList Ontology, and (c) an illustrative example of a project and its ordered releases.

25 http://smiy.sourceforge.net/olo/spec/orderedlistontology.html

4.2.3.3 Modeling Version Ranges

Problem. Manually upgrading dependencies is a tedious work, especially for projects which depend on frequently updated libraries. Version ranges are a measure, supported by several build and dependency management systems, designed to enable developers to automatically upgrade their dependencies without having to adjust the version number in their build file every single time a new version of the dependency is released. However, as introduced in Section 4.2.2, build and dependency management systems define version ranges with different syntaxes.

Solution. Figure 4.7 shows the integration of concepts from Figures 4.5(b) and 4.6(b) to create an effective and flexible model of dependency version ranges in our domain layer. The <sbson:VersionRange> concept uses data properties such as <sbson:exactVersion>, <sbson:lowerThanVersion>, and <sbson:greaterThanVersion> to represent the version range of a dependency link. Details on how the final dependency version is inferred are provide in Section 4.2.3.5.

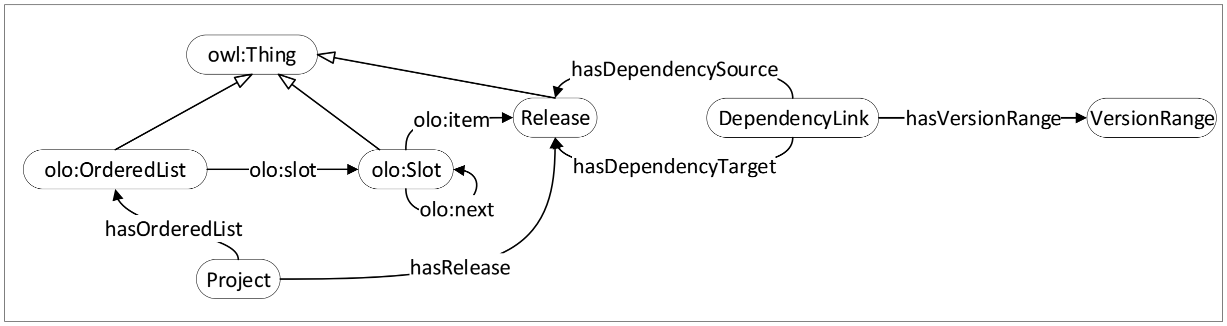


Figure 4.7: Concepts used to model and reason on dependency version ranges

4.2.3.4 Modeling Dependency Exclusions (Filtering)

Problem. Dependency exclusion is a feature provided by most dependency management tools as a way of dependency mediation. It enables users to explicitly exclude specific transitive dependencies when building a project. Such dependency exclusions can occur at two different levels: per-dependency or per-configuration/module. The configuration/module exclusion scope makes it possible to exclude a transitive dependency completely from all dependencies during the project build phase. The per-dependency scope only excludes a transitive dependency for the specified dependency; it is possible that another dependency would re-include that excluded dependency. Ivy and Gradle provide support for both whiles Maven supports per-dependency exclusion. In the scope of this thesis, we focus only on the per-dependency scope.

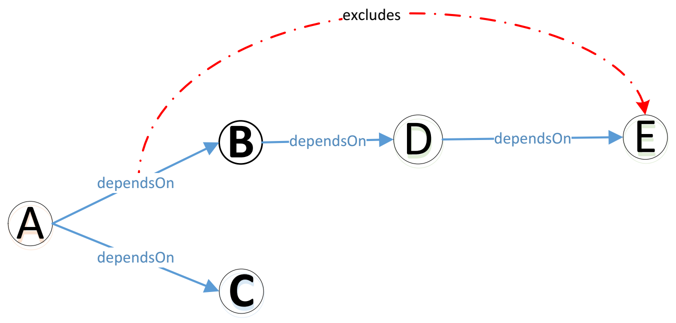


Figure 4.8: Transitive exclusion at per-dependency scope

Figure 4.8 shows an example of a per-dependency exclusion. Project ‘A’ defines a dependency on ‘B’ but excludes the transitive dependency on ‘E’. This means that during the build of ‘A’, project ‘E’ would be excluded from the transitive dependencies of ‘B’. When one wants to query for all transitive dependencies of ‘A’, the result should be {B, C, and D}. Since exclusions are dependency specific, the query results should be project specific. For example, querying the transitive dependencies of ‘B’ should give {D and E} because ‘E’ is not excluded in any of B’s dependency definitions.

Solution. Like the approach used in modeling dependency version ranges, we again use <sbson:DependencyLink> concept from the property reification pattern (see Figure 4.5(b)) to define any dependency-level exclusions on projects or releases through the <sbson:excludesProject> and <sbson:excludesRelease> properties. Figure 4.9 shows an overview of the concepts and relationships involved in this refined model design.

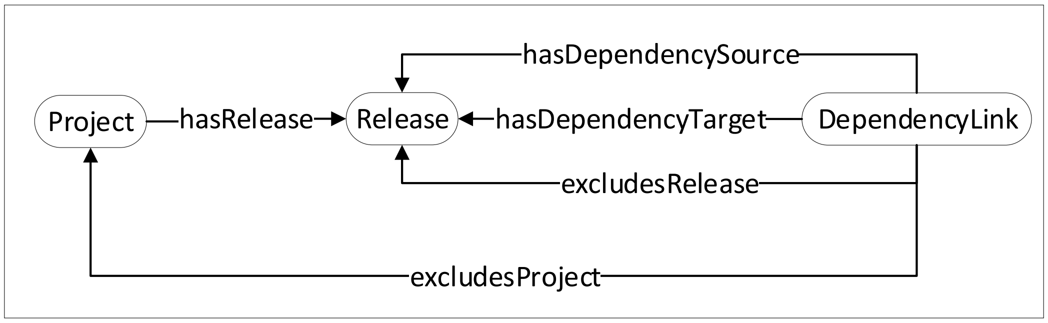


Figure 4.9: Concepts used to model and reason on dependency exclusion

In what follows, we describe how direct and transitive dependencies that can be inferred using the dependency version ranges and exclusions design we have presented so far.

4.2.3.5 Reasoning on Direct and Transitive Dependencies

Problem. As discussed in Section 4.2.3.3 and 4.2.3.4, traditional build and dependency management systems allow developers to specify a version range for direct dependencies and exclude unwanted transitive dependencies. This possibility of version ranges and excluded transitive dependencies in a project's definition makes the automatic resolution of direct and transitive dependencies a non-trivial task. Our modelling approach, using semantic rules and ontology design patterns, offloads much of this challenge (reasoning about dependency resolution) to the SW reasoners. However, as introduced in Section 4.2.3.1, modelling dependency links as an OWL class instead of a property removes the standard support for transitive reasoning on dependencies.

Solution. We introduce custom SWRL rules which take advantage of the scalable reasoning services (e.g., RDFS, RDFS++) provided by the SW stack to reason on dependency resolution. To distinguish between direct and inferred transitive dependencies, we introduce two new properties, <sbson:hasDirectDependencyOn> and <sbson:hasTransitiveDependencyOn>. In what follows, we describe in detail the rules we created that allow us to reason on direct dependencies based on version ranges, and transitive dependencies.

Direct Dependency Reasoning. To allow for the automatic resolution of direct dependencies in our approach, we define three (3) rules which infer the correct instance of a dependency version to assign as the range of the <sbson:hasDirectDependencyOn> property. The rules are based on the following version ranges: exact versions (Figure 4.10), versions lower than a specified value (Figure 4.11), and versions greater than a specified value (Figure 4.12). The rules take advantage of the ordered list pattern (see Figure 4.6(b)) and the dependency link reification pattern (see Figure 4.5(b)) to allow for the inference of the final dependency version to be used.

Transitive Dependency Reasoning. The goal of this reasoning service is to provide flexible and scalable inference of transitive dependencies in the absence or presence of dependency exclusions. Since SWRL does not allow for Negation as Failure[26], rules such as *Person(?p)* ^ ¬

[26] https://github.com/protegeproject/swrlapi/wiki/SWRLLanguageFAQ#Does_SWRL_support_Classical_Negation

hasCar(?p, ?c) → CarlessPerson(?p) are not allowed. Only individuals with an explicit OWL axiom stating that they have no car can be safely concluded to be without a car: *Person(?p) ^ (hasCar = 0)(?p) → CarlessPerson(?p)*. Therefore, to reason about the absence or presence of dependency exclusions, a <sbson:hasNumberOfExclusions> data property is assigned to the <sbson:DependencyLink> concept to store the total number of excluded dependencies defined for a given project dependency. Using this property, our rules in Figures 4.13 and 4.14 infer transitive dependencies in both the presence and absence of exclusions, respectively.

```
DependencyLink(?link),
hasDependencySource(?link, ?release1),
hasDependencyTarget(?link, ?project2),
hasVersionRange(?link, ?range),
exactVersion(?range, ?version),
hasRelease(?project2, ?release2),
hasVersionNumber(?release2, ?version)
→ hasDirectDependencyOn(?release1, ?release2).
```

Figure 4.10: Inferring DirectDependencyOn based on an “exact” version range

```
DependencyLink(?link),
hasDependencySource(?link, ?release1),
hasDependencyTarget(?link, ?project2),
hasVersionRange(?link, ?range),
lowerThanVersion(?range, ?version),
hasRelease(?project2, ?release),
hasVersionNumber(?release, ?version),
hasOrderedList(?project2, ?list),
slot(?list, ?slot1),
slot(?list, ?slot2),
item(?slot1, ?release),
item(?slot2, ?release2),
index(?slot1, ?index1),
index(?slot2, ?index2),
swrlb:substract(?index2, ?index1, 1),
→ hasDirectDependencyOn(?release1, ?release2).
```

Figure 4.11: Inferring DirectDependencyOn based on a “lower than” version range

```
DependencyLink(?link),
hasDependencySource(?link, ?release1),
hasDependencyTarget(?link, ?project2),
hasVersionRange(?link, ?range),
greaterThanVersion(?range, ?version),
hasRelease(?project2, ?release),
hasVersionNumber(?release, ?version),
hasOrderedList(?project2, ?list),
length(?list, ?len),
slot(?list, ?slot1),
slot(?list, ?slot2),
item(?slot1, ?release),
item(?slot2, ?release2),
index(?slot1, ?index1),
index(?slot2, ?len),
swrlb:greaterThan(?len, ?index1),
→ hasDirectDependencyOn(?release1, ?release2).
```

Figure 4.12: Inferring DirectDependencyOn based on a “greater than” version range

```
DependencyLink(?link),
hasDependencySource(?link, ?release1),
hasDependencyTarget(?link, ?release2),
dependsOn(?release2, ?release3),
hasNumberOfExclusions(?link, 0)
→ hasTransitiveDependencyOn (?release1, ?release3).
```

Figure 4.13: Inferring hasTransitiveDependencyOn in the absence of exclusions

```
DependencyLink(?l),
hasDependencySource(?l, ?r1),
hasDependencyTarget(?l, ?r2),
dependsOn(?r2, ?r3),
hasNumberOfExclusions(?link, ?exclusions),
swrlb:greaterThan(?exclusions, 0),
excludesProject(?l, ?p1),
hasRelease(?p2, ?r3),
owl:differentFrom(?p1, ?p2)
→ hasTransitiveDependencyOn (?r1, ?r3).
```

Figure 4.14: Inferring hasTransitiveDependencyOn in the presence of exclusions

4.2.3.6 A Unified Knowledge Representation

The result of our modeling process is SBSON, which describes knowledge from build and dependency management systems using different levels of modeling abstraction. Figure 4.15 shows the core concepts and object properties of our model. It should be noted that data properties have been omitted to improve the readability of the figure.

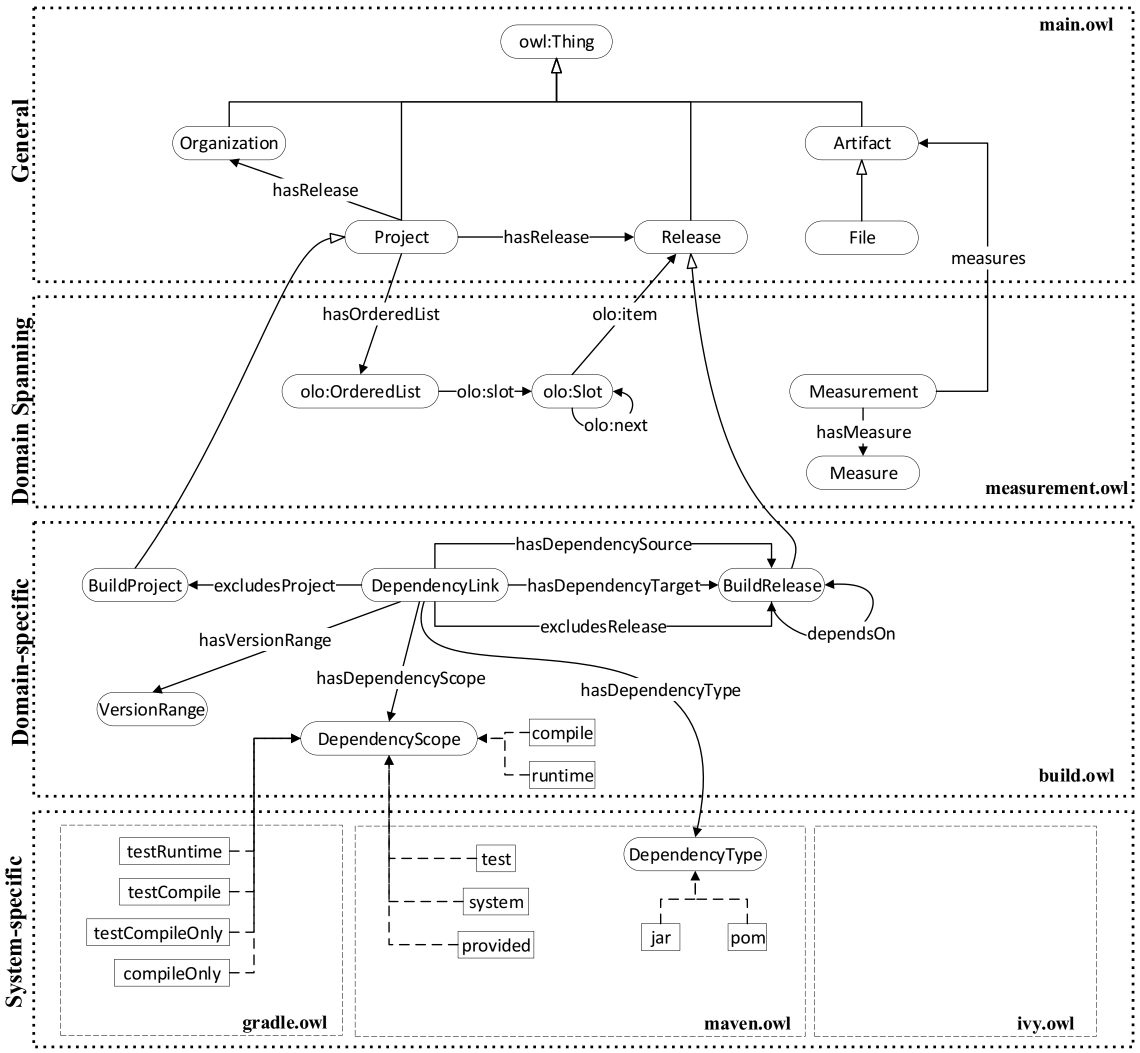


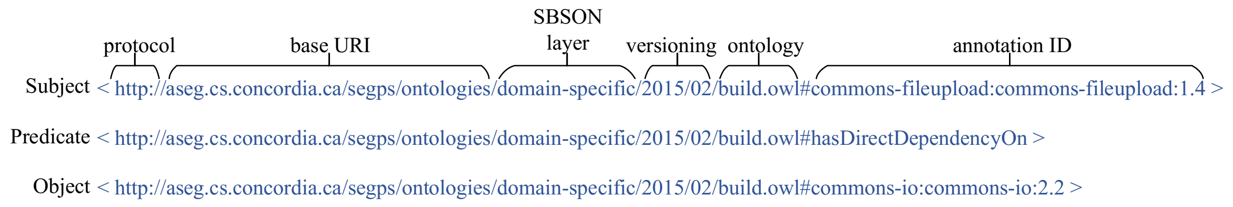
Figure 4.15: Overview of the concepts and (object) properties in the unified SBSON family of ontologies

A key concept in our knowledge model is the <sbson:BuildRelease> (domain-specific level), which is a subclass of the <main:Release> concept (general layer). Build releases model distributed releases of software projects, captured by the <sbson:BuildProject> concept (domain-specific level). These build releases are stored in online build repositories such as Maven Central. Multiple releases of a project are ordered using Slots in an <olo:OrderedList> (domain-spanning level). In our modeling approach, build releases define their dependencies on other releases using a <sbson:DependencyLink> (domain-specific level). Special characteristics of a dependency link are represented using the <sbson:VersionRange> and <sbson:DependencyScope>, both being domain-specific concepts, and the <sbson:DependencyType> concept at the system level. Scope of dependencies, as well as the dependency types are specific to a build system and are therefore modeled in the individual system ontologies.

4.2.4 Step 4: Ontology Population

In this step, we describe how the knowledge extracted from the Maven Central Repository is automatically transformed into semantic triples based on the RDF framework. The transformation and population process relies on the generation of unique, de-referenceable and HTTP-resolvable URIs for the resulting triples.

Figure 4.16 shows an example for a triple that is generated for an instance of a direct project dependency. Each generated URI contains a base URI, followed by the SBSON layer, knowledge version, and ontology to which that fact belongs. This is followed by the annotation ID; the annotation ID identifies whether a given URI represents a semantic type (e.g., hasDirectDependencyOn) or a populated individual (e.g., commons-fileupload:commons-fileupload:1.4),



Subject < http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#commons-fileupload:commons-fileupload:1.4 >

Predicate < http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#hasDirectDependencyOn >

Object < http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#commons-io:commons-io:2.2 >

Figure 4.16: Anatomy of the URI of a generated triple

4.2.5 Step 5: Ontology Evolution

The last step of our methodology reflects the fact that that our knowledge modelling approach is an iterative process, with our ontologies evolving as additional build and dependency management systems are included in our knowledge model. The addition of new system-specific ontologies can lead to changes in the existing abstracted domain ontology. In addition to the inclusion of new concepts and properties to the domain ontology to capture new dependency management features and semantics, there is the possibility that existing domain concepts and properties will be demoted to the lower layer. As discussed earlier, a key benefit of using ontologies is that they are can be extended as relationships and concept matching are easy to add to existing ontologies. As a result, ontologies can evolve with the growth of data without impacting dependent processes and systems if something goes wrong or needs to be changed.

4.3 Chapter Summary

In this chapter, we presented an approach for developing an ontology-based knowledge model for build and dependency management systems (SBSON). Our approach allows us to reconcile and integrate heterogeneous build system facts from several build systems. Our knowledge modeling approach takes advantage of OWL reasoning capabilities as well as existing ontology design patterns to abstract and reuse concepts across system level ontologies, while at the same time improving knowledge integration and reuse.

In the following chapters we describe how we integrated SBSON with other SE knowledge sources (e.g., source code, vulnerability databases, version control, and software licenses) to provide a comprehensive, interlinked knowledge base that allows for novel types of analysis across artifact and project boundaries.

Chapter 5

5 A Semantic Web Enabled Approach for the Early Detection of API Breaking Change Impacts

5.1 Introduction

Global code reuse, through libraries and components developed by third-parties or the open source community, has become an essential aspect of today's software development processes [2], [16]. However, as these libraries evolve to accommodate new features, bug fixes, and general code improvements, some of these library changes (often referred to as breaking changes) may break already established contracts, leading to errors and requiring rework in client applications. These library incompatibilities can also cause a ripple effect within a software ecosystem, requiring code changes in dependent client projects within the ecosystem to mitigate the impacts of these breaking changes.

For library consumers, analyzing direct dependencies used in a project reveals often only the tip of the iceberg; most of the complexity and challenges in analyzing breaking changes are caused by transitive dependencies. For example, in Figure 5.1, since P1 (unknown to the developer) depends on multiple versions of P2, there is a potential of unexpected runtime failures if the versions of P2 are backward-incompatible. Being aware of who and how client projects use their API can help library producers make better choices when dealing with breaking changes (e.g., API producers of P2 can analyze the potential impact of API changes on clients such as P1 and P6 prior to committing to these changes).

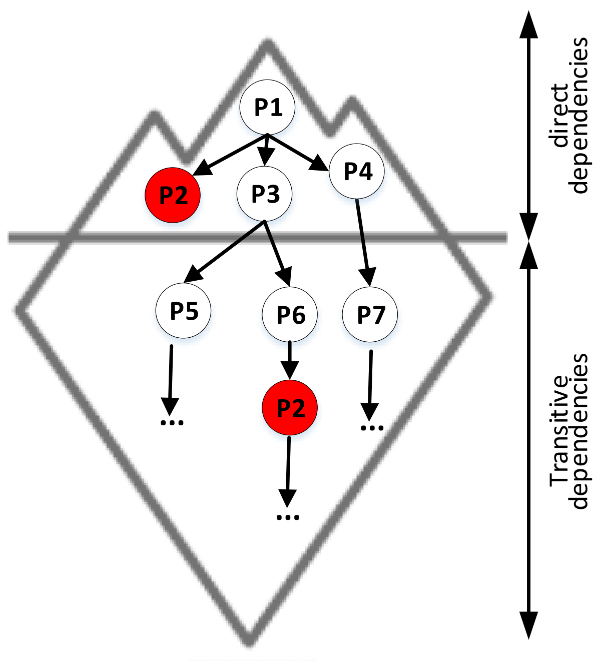


Figure 5.1: The hidden complexity of breaking changes due to transitive dependencies

Previous studies have shown that breaking changes are frequent, and that many library producers and consumers are aware of this fact [99]–[102]. Several solutions have been proposed to help mitigate the impact of breaking changes. From a library producer's perspective, different strategies can be adopted to either shift or delay the cost associated with API changes. A study by [18] on the Eclipse, Node.js/npm, and R/CRAN ecosystems identified four major strategies: maintaining old interfaces, having parallel releases, release planning, and communication with users. The strategy adopted by individual ecosystems is dependent on its overall development objective and the cost associated with applying a mitigation strategy. For example, in the Eclipse community, developers prefer to incur higher cost to maintain Eclipse's backward-compatibility with older releases; developers can extend interfaces by creating new interfaces and deprecate older interfaces without having to remove them [18]. However, each mitigation strategy has its own challenges, such as incurring additional maintenance overhead or introducing opportunity costs (technical debt). Also, the potential impact of a breaking change on library consumers plays an important factor when deciding on a mitigation strategy.

Similarly, different approaches for consumers of software libraries have been introduced to support library migration. These approaches are based on comparing two library versions (e.g., using lexical comparison of method signatures [99], [103]). In case there are differences, attempts can be made to reconcile these differences by comparing if the new functionality in a library is a code clone of previous functionality [104]; identifying how a library's use of its API has changed [105]; identifying how other developers have migrated their code or test suites to accommodate the API change [106]; and using a combination of these techniques [99]. Unfortunately, these reconciliation approaches neither allow library producers to assess the impacts of a chosen mitigation strategy on their clients, nor do they provide developers of client applications with any support in assessing the potential risks and effort involved in their migration tasks.

In our research, we introduce a novel approach to support API consumers in identifying the potential impacts of an API change on their product and for API producers in managing (assessing) the impacts of breaking changes on other projects or a complete ecosystem. More specifically, we take advantage of the Semantic Web and its technology stack (e.g., ontologies, reasoning services) to establish a unified knowledge representation that supports the analysis of both direct and indirect (transitive) third-party library usage across thousands of open source projects. Much of the flexibility of our approach is based on the use of inference services provided by the Semantic Web to infer explicit and implicit knowledge from the knowledge base. To evaluate the effectiveness of our approach in identifying the impact of breaking changes on project dependencies and complete ecosystems, we conducted a case study on Java projects available in the Maven ecosystem. Nonetheless, it should be noted that our approach is independent of the type of dependency management or programming languages being used.

5.2 Background

5.2.1 API Usage and Breaking Changes

Software libraries take advantage of visibility modifiers (e.g., public and protected in Java) to provide reusable and extendable APIs to other applications. However, as these software libraries evolve, changes made to their APIs might impact many external clients. Such changes are classified into breaking and non-breaking changes [107] as follows:

- Breaking changes break backward compatibility through removal or modification of API elements. Consequently, clients may face compilation errors after updating. Common examples include removal of classes or methods, visibility loss (e.g., public to private), and changes to a method's return type or parameters.

- Non-breaking changes preserve compatibility among interfaces and usually involve the addition of new functionalities to the library. Examples include visibility gain (e.g., private to public) and deprecated method removal.

Although performing a change to a library might be a straightforward task, resulting breaking changes can have a significant ripple effect on client projects depending on how the changed API is used throughout their project. Certain API usages expose client projects to API changes more than others [108]. Wu et al. [108] categorized API usages into API-injection usages and local usages. API-injection usages occur when a library's API becomes part of a client project through inheritance, interface implementation, and using reference types as method return types or parameter types. For local API usages a library's API is used within the body of a method. As a result, such breaking changes in a library's API will require changes to any client method that directly or transitively uses a modified API.

5.2.2 Software Evolution ONtologies (SEON)

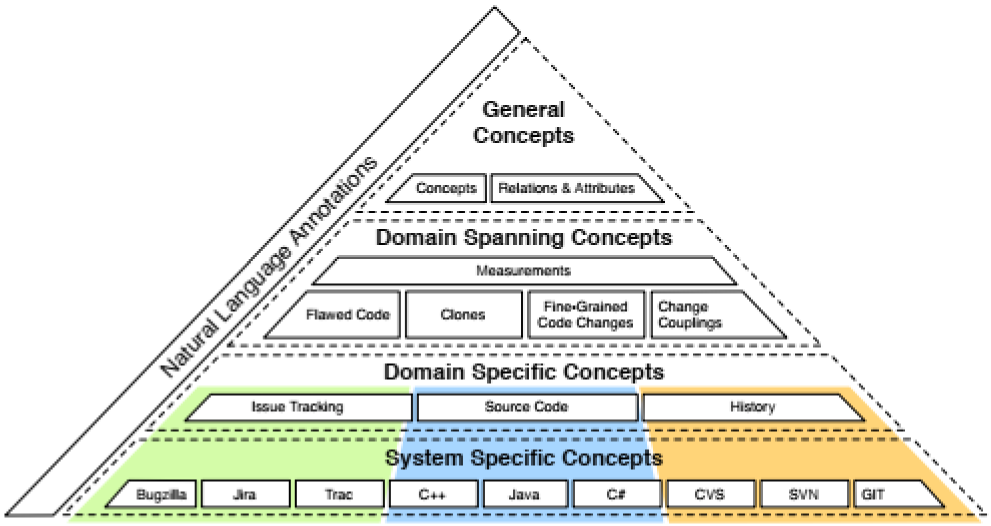


Figure 5.2: Overview of the SEON pyramid of ontologies [34]

The Software Evolution Ontologies (SEON), introduced by [34], provides a shared taxonomy of important software engineering concepts and demonstrates how software evolution knowledge can be adequately represented by means of ontologies. SEON establishes a shared taxonomy for explicitly describing relationships among artifacts, and for linking data such as code structures, issues (change requests), bugs, and basically any changes made to a system over time. SEON is constructed following a bottom-up approach which iteratively abstracts concepts found in common software evolution analysis and tools into different layers. Figure 5.2 presents an overview of the different layers of SEON.

5.3 A User Survey on the Impact of API Breaking Changes

Prior work on API breaking changes has examined different migration techniques available to API consumers (e.g., [103]–[106]). However, these existing techniques do not address how consumers can assess the risk or effort involved when selecting a migration technique or how producers can analyze potential impacts of their library changes on a global software ecosystem. To gain a better understanding on how API consumers and producers currently deal with the impact of breaking changes, we conducted a survey involving open source developers. Survey participants were identified by mining developer emails from publicly available projects hosted on Maven and GitHub. We manually cleaned the list of emails to ensure that it did not contain any educational or organizational emails. We sent an invitation to 1000 randomly selected participants from the collected e-mail addresses and received a total of 54 responses, i.e., an acceptable response rate of 5.4%, which is in line with response rates reported by other software engineering surveys [109]). From, these 54 responses, we excluded one response from our further analysis, due to the explicit request of the survey participant, leaving us with 53 survey participants.

Survey Design: We designed an online survey that included four main parts. First, we asked questions related to a participant's background and experience. We also asked participants about their preferred development ecosystems, the roles they play within it, and their experience with breaking changes as either client or producer of software libraries (Table 5.1). Finally, we solicited their views on features which they would consider essential in helping developers to identify and manage the impacts of breaking changes.

Of the 53 survey participants, 42 participants had more than 5 years of development experience, 10 respondents had between 1 to 5 years, and 1 responder had less than 1 year of experience. 23 participants identified Maven as their preferred ecosystem, 11 participants selected NPM, 7 participants chose PyPi, 2 selected Packagist, and 10 participants listed other ecosystems.

As for their role in these ecosystems, 24 participants identified themselves as core contributors of software libraries, 22 participants as consumers of libraries, and the remaining 7 participants have contributed patches to open source libraries. Overall, the participants are quite experienced in software development and the use of open source libraries/packages.

Table 5.1: Background of survey participants

Experience (in years)	**#**	**Ecosystem Used**	**#**	**Ecosystem Role**	**#**
< 1	1	Maven	23	Library consumer	24
1 – 2	1	NPM	11	Core contributor	22
2 – 5	9	PyPi	7	Patch submission	7
5 – 10	18	Packagist	2		
> 10	24	Other	10		

5.3.1 How often do developers experience breaking changes?

We asked developers to share their experience on how they have been affected in the past by breaking changes (Table 5.2). For direct breaking changes, 55.1% of the participants indicated they experience them several times a year and 2% of the participants deal with them several times a month. The remaining 42.8% indicated that they were rarely or never exposed to direct breaking changes. In a second question, we asked the participants regarding breaking changes within transitive dependencies. Among the survey participants, 55.1% have been exposed to breaking changes due to transitive dependencies, 12.2% of responders were never impacted, and the remaining 32.7% were unsure.

Table 5.2: Report on breaking changes experienced by developers

Frequency of direct impacts	**%**	**Experienced indirect impacts**	**%**
Several times a month	2	Yes	55.1
Several times a year	55.1	No	12.2
< once a year	36.7	Maybe	32.7
Never	6.1		

We further asked the participants to describe in an open-ended question, how they discovered that a breaking change occurred in their project. 19 out of the 53 participants responded to this question. Failing tests and builds (31.6%) and runtime exceptions (26.3%) were the most common indicators used for detecting breaking changes caused by transitive dependencies. In some cases, the impacts of the breaking changes were only discovered when an application was already in production. For example, participant P48 stated that: "*We had to update a transitive dependency pulled in by some library due to a security vulnerability. After the update, PDF and other document formats indexing stopped working. We realized that only in production and took some debugging to discover and fix the cause. A unit test was written alongside the fix in order to mitigate a possible reoccurrence*".

5.3.2 What features would developers need for identifying and managing the impacts of breaking changes?

We further asked participants what features they would find useful in helping them to identify and manage the impact of these changes. Responses from API consumers show that better support for the impact analysis of breaking changes is needed. Among the most striking responses are the following statements by participants P17 and P22, describing the need to provide better support for API consumers: P17 said, *"... I'd like to see that I'm using the incompatible interfaces*", and P22 "*Version compatibility. Whenever a new library is added to an existing system, developers don't know about the impact caused by this library with the other libraries which are already in the system [...]*". API producers such as participant P11 stated: "*Prevent releases from making to production unless confirmed by software owners that it's safe*." Overall, the responses from our survey participants indicate that developers would like to see additional tool support for detecting and analyzing the impact of breaking changes, which leads us to formulate the following research questions:

- ***RQ1***: For library developers, can knowledge on how their APIs are used by client projects be useful for the selection of a breaking change mitigation strategy?
- ***RQ2***: Can our approach identify incompatibilities within transitive library dependencies which might lead to unexpected runtime behavior?

5.4 Modeling the Impact of API Breaking Changes

5.4.1 Modeling and Integration of the Source Code Ontology

A fundamental premise of the Semantic Web is its ability to share and extend existing knowledge. Our knowledge modeling approach builds upon this premise by reusing and extending existing software engineering ontologies introduced in [34].

Since some of our proposed API impact analysis requires access to source code information, we introduced our Source COde ONtologies (SOCON) which is an extension of SEON's domain-level source code ontology [34]. SOCON introduces additional concepts and properties to model API breaking changes and their impact. In addition, we introduced the <code:containsCodeEntity> property, and its inverse <code:foundInRelease> property, to link project releases in SBSON to their internal code elements in SEON. Using an ontological knowledge representation for both build repositories and source code allows for a seamless semantic integration of both knowledge resources. In addition, it allows us to take advantage of reasoning services provided by the Semantic Web to infer new knowledge. In what follows, we present in more details our knowledge model and how this model takes advantage of the Semantic Web inference services.

Figure 5.3 summarizes the main concepts and object properties, found in the four abstraction layers of our model used for the impact analysis of API breaking changes. It should be noted that data properties have been omitted to improve the readability of the figure. Also, we use prefixes as substitutes to the fully qualified names of our ontologies (the prefixes can be dereferenced using the URIs shown in Appendix A).

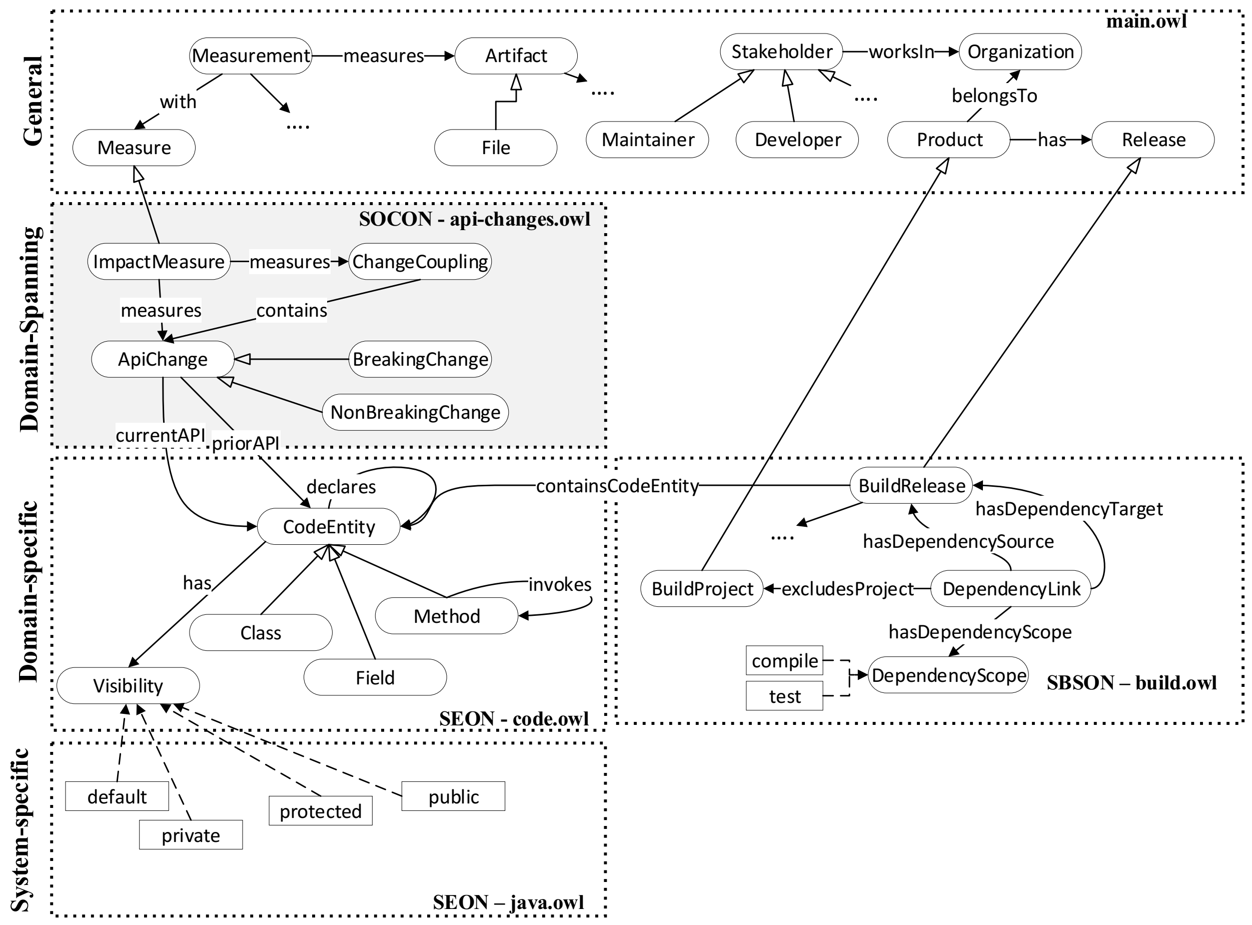


Figure 5.3: Ontologies and concepts involved in API change impact analysis

An important concept in our knowledge model is the <sbson:BuildRelease> concept (located at the domain-level of our SBSON ontology), which is a subclass of the Release concept found in the general SBSON ontology layer. <sbson:BuildRelease> models distributed releases of software projects, where a build release isReleaseOf a <sbson:BuildProject>, and models that a project can have several releases. <sbson:BuildRelease> defines its dependencies on other releases using <sbson:DependencyLink>. Stakeholders, such as Developers, distribute new releases which can lead to <code:ApiChanges>.

An API change can either be a <code:BreakingChange> or a <code:NonBreakingChange>. API changes are detected by comparing <code:CodeEntity> individuals using the <code:priorAPI> and <code:currentAPI> relations. Code changes are captured at different entity levels such as <code:Field>, <code:Method>, and <code:Class>. A <code:ChangeCoupling>

contains all API changes which coexist due to a dependency between API elements. Furthermore, our domain level ontology for source code includes a <code:Visibility> concept. In most object-oriented programming languages, there exists a mechanism for information-hiding exists to control the access to parts of the code (e.g., in Java public, default, protected, and private are used to specify the visibility of methods and fields). These visibility modifiers are defined in the system-specific (Java) ontology since the semantics of visibility modifiers might vary among programming languages. For a complete description of our ontologies, we refer the reader to [98].

5.4.2 Knowledge Inferencing and Reasoning

As previously mentioned, the Semantic Web stack provides support for scalable inference on big data through its reasoning services (e.g., RDFS, RDFS++). In our work, we take advantage of these reasoning services to support different types of dependency analysis and to infer new knowledge. In addition, these reasoning services in combination with user-defined queries, allow us to replace traditionally proprietary graph and tree traversal implementations used by existing analysis tools. In what follows, we describe in more detail how our knowledge model supports transitivity and subsumption reasoning.

Transitive closure inference: In mathematics, the transitive closure of a binary relation R on a set X is the smallest relation on X that contains R and is transitive[27]. For example, if X is a set of class methods and x R y, then method x invokes method y. The transitive closure of R on X is therefore the relation R+ such that x R+ y, reflecting that method x can call method y through several indirect method invocations. Such transitive dependencies can be expressed in OWL through the <owl:TransitiveProperty> construct, which we use to define <code:invokesMethod> as a transitive property. This transitive property allows us to retrieve a list of all direct or indirect invocation dependencies for a given method and vice-versa (Figure 5.4).

[27] https://en.wikipedia.org/wiki/Transitive_closure

```
SELECT ?method
WHERE {
  ?method rdf:type code:Method.
  ?method code:invokesMethod <subjectMethodURI> option(transitive).
}
```
Figure 5.4: SPARQL query returning transitive method calls

Subsumption inference: Another essential aspect of our ontology design is its support for subsumption hierarchies between its concepts [110]. For example, a <code:Method> or <code:Class> is a sub-concept of <code:CodeEntity>. Subsumption hierarchies add significant power to ontologies [33] by comparing the syntactic structure of concept descriptions. Given a set of concepts C, the goal of the inference engine is to discover all subsumption relationships among pairs of concepts in C. More formally, we can denote that concept c1 is a sub-concept of c2 by c1 ⊆ c2. Subsumption is directional [110]: if c1 ⊆ c2, then c2 !⊆ c1 unless c1 and c2 are synonyms. Similar subsumption can be inferred from OWL properties that can subsume each other.

In our modeling approach, we support subsumption reasoning by creating a hierarchy of object properties that capture source code dependencies (Figure 5.5).

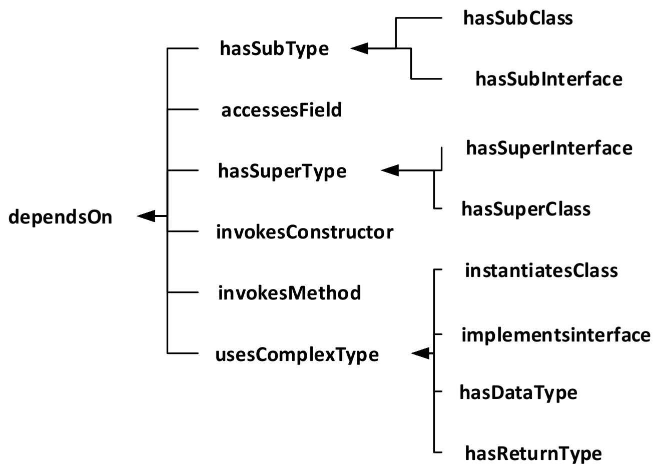


Figure 5.5: Hierarchy of code properties

Using the SPARQL query (Figure 5.6), which combines the property hierarchy with subsumption inference, all code entities that transitively depend on another code entity, independent of their types (e.g., method invocations, interface implementation), can be identified.

```
SELECT ?entity
WHERE {
 ?entity rdf:type code:CodeEntity.
 ?entity main:dependsOn <subjectEntityURI> option(transitive).
}
```

Figure 5.6: Query illustrating the dependsOn subsumption inference

5.5 Case Study

The overall objective of this section is to illustrate the applicability and flexibility of our knowledge model. More specifically, we show how are approach can help both library developers and consumers in identifying the impacts of breaking changes within and across project boundaries.

5.5.1 Dataset Description

For our case study, we take advantage of the Maven ecosystem as our primary dataset, since the repository hosts many popular and widely used open source libraries. The Maven repository, like other repositories used by build management tools, includes structured dependency and version information, which are required for performing API change and usage analysis.

More specifically, for identifying the impacts of breaking changes, we selected the ASM[28] library, a Java bytecode manipulation library, which is hosted on Maven. We selected ASM for our case study, since the library underwent a radical redesign from release 3.X to 4.0. As part of this redesign, Release 4.0 introduced several breaking changes (e.g., interfaces were changed to abstract classes, breaking previous 3.X API versions). Table 5.3 and 5.4 summarize the details of our Maven and ASM datasets.

Table 5.3: Summary of Maven dataset

# Projects	**# Releases**
130895	1,219,731

[28] http://asm.ow2.org/

Table 5.4: Summary of ASM dataset

ASM Project	# Releases	# Unique Dependencies
ASM 3.X and older	20	364
ASM 4.X and newer	13	848

5.5.2 Results

In what follows, we report on the results of our case study, which includes for each research question, the motivation, the approach being used, and our findings.

RQ1: For library developers, can knowledge on how their APIs are used by client projects be useful for the selection of a breaking change mitigation strategy?

Motivation: As discussed in the introduction of this chapter, library producers adopt different strategies to either shift or delay the cost associated with making API changes. For library producers to decide on a mitigation strategy, they have to be able to determine the potential impact of their API changes on other projects. In particular, since some breaking API changes (e.g., interface implementation, inheritance, and using reference types as return or parameter types) require a significant effort from API consumers to modify their application design and implementation. To reduce the potential impact of these API changes for consumers, library producers should be aware of the use of their API in client programs. Being aware of such potential impacts of API changes on client systems can guide API producers in selecting an appropriate mitigation strategy that reduces the potential impact of such changes to consumer systems.

Approach: Figure 5.7 shows the overall methodology we used for our case study, which includes extracting and populating facts for breaking changes between different ASM releases (using VTracker[29]), source code of ASM releases and their dependencies, and the complete dependency information of the Maven repository. Using the subsumption inference (see Section 5.4.2), we can now identify how client projects use the changed ASM APIs. We use the query in Figure 5.8 3 to extract these different usage types of a given API within a client project.

[29] https://users.encs.concordia.ca/~nikolaos/vtracker.html

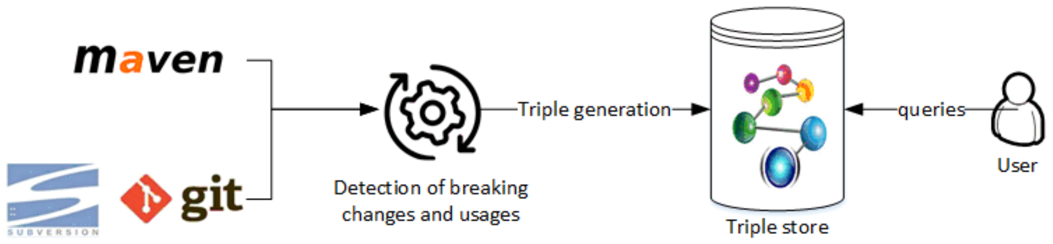


Figure 5.7: Overview of approach for breaking change impact analysis

```
SELECT ?client ?clientEntity ?usageType
WHERE {
  ?client main:containsFile ?clientFile.
  ?clientFile code:containsCodeEntity ?clientEntity.
  ?clientEntity ?usageType <subjectAPI>.
}
```

Figure 5.8: SPARQL query to identify API usage in client projects

Findings and Discussion: Table 5.5 summarizes the client usages (internal and external) of selected ASM APIs for which we identified breaking changes. As our results show, interface implementation and class inheritance are among the most common types of API usage types in client projects. In contrast, library producers used their own libraries mostly as return types. This highlights that library producers and consumers not only use APIs differently, but also selecting a breaking change mitigation approach based on the internal API usage is often not enough. As the case study illustrates, our modeling approach supports both internal and cross-project breaking change and API usage analysis. This additional insight on potential global impacts of breaking API changes, can guide API producers to determine potential mitigation strategies (e.g., deprecation) and therefore, maintaining their API's value proposition – the reuse of functionalities, while minimizing development, maintenance and testing effort required by the consumers.

Table 5.5: Summary of External and Internal Usage of selected ASM APIs

API	API Usage Type	# Client (external) Usages	# Internal Usages
ClassVisitor	Implement interface	86	6
	Return type	0	24
ClassAdapter	Inherit class	44	0
	Return type	0	24
MethodVisitor	Implement interface	4	4
	Return type	0	47
MethodAdapter	Inherit class	2	0
	Return type	0	47

RQ2: Can our approach identify incompatibilities within transitive library dependencies which might lead to unexpected runtime behavior?

Motivation: As introduced in Section 4.2.1, different build and dependency management systems adopt different conflict resolution techniques to deal with multiple versions of a dependency in a project. For example, Maven chooses the version of the dependency closest to the root of the dependency tree. However, this type of conflict mediation, can lead to potential runtime failures, which are not identified during the build or compilation process. Being able to identify the use of changed APIs across transitive dependencies allows library consumers to avoid some of these unexpected runtime failures.

Approach: As described in Section 5.4.1, the concepts <code:CodeEntity> and <code:BreakingCodeChange> are used to represent source code syntax and its semantics. <sbson:DependencyLink> using the <sbson:hasDependencyType>, <sbson:hasDependencyScope>, and <sbson:hasDependencyExclusion> properties capture the dependency between two software libraries. Based on this representation, developers can now use (user and predefined) SPARQL queries to analyze whether their application is exposed to potential direct and indirect breaking changes. For example, the query in Figure 5.9 identifies all projects that are dependent (either direct or transitive) on different versions of the ASM library.

Although ASM versions may contain binary incompatibilities, the inclusion of these APIs in a client project's build might not automatically result in breaking changes. For these changes to become breaking changes, an incompatible API must be invoked. For our analysis, we create therefore a static, global call graph to determine if a changed API is (potentially) called by the client application.

```
SELECT ?project ?asm1 ?asm2
WHERE {
 <…/build.owl#org.ow2.asm:asm> main:hasRelease ?asm1.
 <…/build.owl#org.ow2.asm:asm> main:hasRelease ?asm2.
 ?project build:hasDirectDependencyOn ?asm1.
 ?project build:hasTransitiveDependencyOn ?asm2.
 FILTER(?asm1 != ?asm2).
}
```

Figure 5.9: SPARQL query identifying the use of multiple versions of the ASM library in projects

```
SELECT ?client ?clientAPIEntity2 ?dependency ?dependencyAPIEntity
WHERE {
  #identify use of breaking change entity in client and dependency
  ?client code:containsCodeEntity ?clientAPIEntity1; code:containsCodeEntity ?clientAPIEntity2.
  ?clientAPIEntity1 main:dependsOn ?currentAPIElement.
  ?dependency code:containsCodeEntity ?dependencyAPIEntity.
  ?dependencyAPIEntity main:dependsOn ?priorAPIElement.
  ?clientAPIEntity2 main:dependsOn ?dependencyAPIEntity.
  {
    SELECT ?client, ?dependency ?asm1, ?asm2
    WHERE {
      <…/build.owl#org.ow2.asm:asm> main:hasRelease ?asm1; main:hasRelease ?asm2.
      ?client build:hasDirectDependencyOn ?asm1; build: hasDirectDependencyOn ?dependency.
      ?dependency build: hasDirectDependencyOn ?asm2.
      #Identify ASM releases for which breaking changes have been populated in the KB
      ?breakingChange a code:BreakingCodeChange; code:hasPriorAPI ?priorAPIElement.
      ?breakingChange code:hasCurrentAPI ?currentAPIElement.
      ?asm1 code:containsCodeEntity ?currentAPIElement.
      ?asm2 code:containsCodeEntity ?priorAPIElement.
      FILTER(?asm1 != ?asm2)
    }
  }
}
```

Figure 5.10: SPARQL query to identify transitive usage of API elements impacted by breaking changes

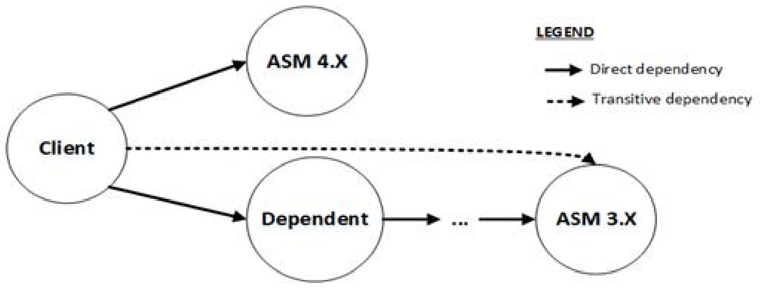


Figure 5.11: Illustrative example of a client project using different versions of the same API

In what follows, we refer to a client as all projects which have declared a dependency on any ASM 4+ library; dependent refers to projects (directly used by a client) which have a dependency to an ASM library version 3.X or older (see Fig. 5.11). The query in Figure 5.10 (an extension of Figure 5.9) returns such transitive usages of different API versions within a project. The query first identifies two unique ASM releases that contain breaking changes and then identifies any usage of these incompatible APIs within client projects and their transitive build dependencies.

Findings and Discussion: The boxplots in Figure 5.12 summarize the distribution of dependents among clients as well as the usage of potential incompatible APIs within the client and dependent projects. Clients, on average, include 5 dependents which may introduce different versions of the ASM library as part of their classpath. Our analysis (Figure 5.10) further shows that 0.21 of ASM API 4.X functionality is invoked on average by a client and 0.34 of functionality from an earlier ASM version (ASM 3.X or earlier) is invoked by a dependent.

Table 5.6 provides two concrete examples where clients are exposed to potential runtime errors due to their indirect use of incompatible ASM API versions. The solr-shade 2.0.0 project directly depends on ASM v4.1 and indirectly on ASM v3.1, since lucene-expressions 4.7.1 which is used by solr-shade 2.0.0, depends on ASM v3.1.

Using Maven’s built-in conflict mitigation, ASM v3.1 will automatically be excluded from the project, and only ASM v4.1 will be included. As a result, an unexpected runtime exception will be thrown when the fromExpression method, which indirectly invokes the now excluded ASMv3.1 ClassVisitor and MethodVisitor APIs.

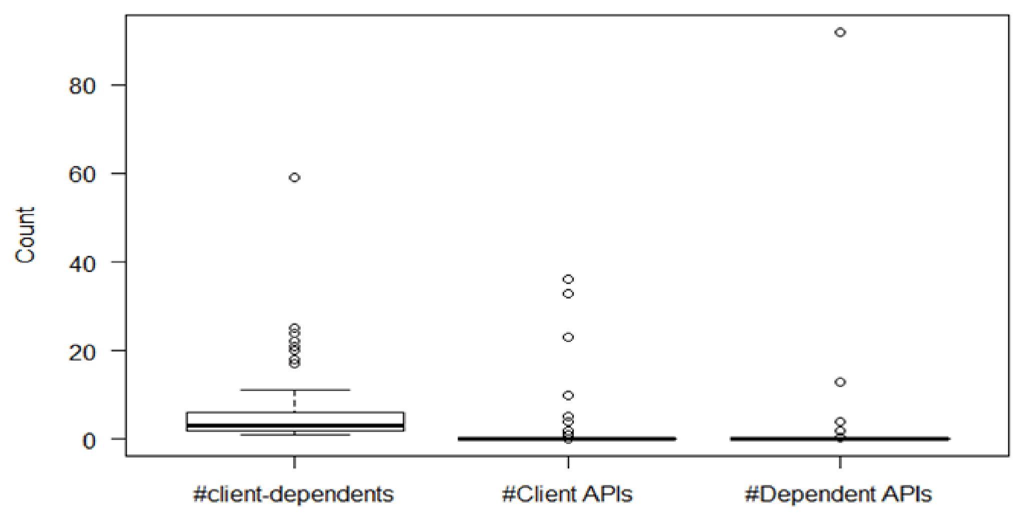


Figure 5.12: Distribution of client dependencies and their usage of incompatible ASM APIs

Table 5.6: Results of potentially impacted Client Projects

Client Project	Potentially Impacted API
solr-shade 2.0.0	Class: DocumentExpressionDictionaryFactory Method:fromExpression(String, Set<org.apache.lucene.search.SortField>)
lucene-expressions 6.0.1	Class: JavascriptCompiler Method: compileExpression(ClassLoader)

In order to evaluate whether our approach can correctly identify the impact of breaking changes, we conducted an additional evaluation. For this study, we used already closed issues, which we extracted from GitHub. These issues contain information in their bug description, indicating whether a bug was due to a breaking change. For dataset selection, we search GitHub for issues that contain certain Java runtime exceptions. These exceptions have previously been reported to be frequently caused by breaking changes [111]: ClassNotFoundException, NoSuchMethodError, IncompatibleClassChangeError, and NoClassDefFoundError.

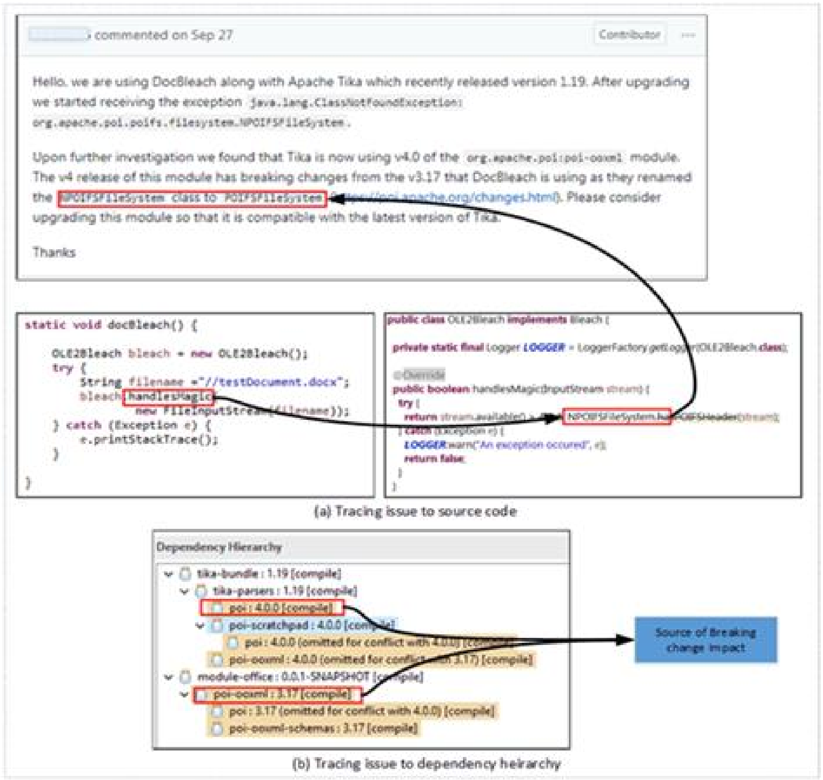
For our dataset, we initially selected the top 800 results (200 results for each keyword), from which only 396 issues used Maven as their build and dependency management tool. As part of our data cleaning, we manually analyzed these remaining 396 results based on the following two processing criteria: (1) we only consider issues that are directly related to breaking changes in the project dependencies and (2) only include projects that successfully build using their default configurations. After applying this pre-processing, only 10 issues remained in our dataset, which are summarized in Table 5.7. For each issue, we downloaded a snapshot of the source code prior to the resolution of an issue and applied the same processing steps described earlier in Figure 5.7.

As our evaluation (Table 5.7) shows, we were able to successfully reproduce and identify the reported breaking changes for 4 out of the 10 issues. Figure 5.13 shows an example of a trace which we established from the issue to the source code and its dependency hierarchy. For the remaining 6 issues, we were unable to identify the breaking changes for various reasons. The most common reason was that we did not have access to the required third-party AWS and Hadoop services to replicate Issues #2, #3, #5 and #6. Our manual analysis of the remaining two issues (Issues #4 and #7) showed that these breaking changes were incorrectly identified since they occurred in dependencies used by the Maven plugins. Although, we did not include these Maven plugin dependencies in our current analysis, extending our queries to cover these dependencies in our analysis is a straightforward task and will be part of our future work.

The results of our study and evaluation show that our formal knowledge representation allows us to take advantage of transitivity and subsumption inference supported by Semantic Web reasoners. In addition, it also provides us with flexibility in terms of being able to write custom queries that support the analysis of breaking changes across artifacts (e.g., Maven and source code) and project boundaries. Such cross-project impact analysis can help library consumers in identifying potential binary incompatibility errors that are usually only discovered during the execution of their project(s).

Table 5.7: Identified potential breaking changes

ID	**Issue URL**	**Identified**
Iss1	docbleach/DocBleach/issues/39	Yes
Iss2	jenkinsci/artifact-manager-s3-plugin/pull/66	No
Iss3	locationtech/geowave/issues/1371	No
Iss4	mulesoft-labs/raml-for-jax-rs/issues/364	No
Iss5	sakserv/hadoop-mini-clusters/issues/35	No
Iss6	ShifuML/shifu/issues/504	No
Iss7	STAMP-project/dspot/issues/424	No
Iss8	togglz/togglz/issues/282	Yes
Iss9	VanRoy/spring-data-jest/issues/74	Yes
Iss10	VanRoy/spring-data-jest/issues/84	Yes


[redacted] commented on Sep 27 Contributor

Hello, we are using DocBleach along with Apache Tika which recently released version 1.19. After upgrading we started receiving the exception `java.lang.ClassNotFoundException: org.apache.poi.poifs.filesystem.NPOIFSFileSystem`.

Upon further investigation we found that Tika is now using v4.0 of the `org.apache.poi:poi-ooxml` module. The v4 release of this module has breaking changes from the v3.17 that DocBleach is using as they renamed the `NPOIFSFileSystem` class to `POIFSFileSystem` (https://poi.apache.org/changes.html). Please consider upgrading this module so that it is compatible with the latest version of Tika.

Thanks

```
static void docBleach() {

    OLE2Bleach bleach = new OLE2Bleach();
    try {
        String filename ="//testDocument.docx";
        bleach.handlesMagic(
                new FileInputStream(filename));
    } catch (Exception e) {
        e.printStackTrace();
    }

}
```

```
public class OLE2Bleach implements Bleach {

 private static final Logger LOGGER = LoggerFactory.getLogger(OLE2Bleach.class);

 @Override
 public boolean handlesMagic(InputStream stream) {
  try {
   return stream.available() > 0 && NPOIFSFileSystem.hasPOIFSHeader(stream);
  } catch (Exception e) {
   LOGGER.warn("An exception occured", e);
   return false;
  }
 }
```

(a) Tracing issue to source code

```
Dependency Hierarchy
v tika-bundle : 1.19 [compile]
  v tika-parsers : 1.19 [compile]
      poi : 4.0.0 [compile]
    v poi-scratchpad : 4.0.0 [compile]
        poi : 4.0.0 (omitted for conflict with 4.0.0) [compile]
      poi-ooxml : 4.0.0 (omitted for conflict with 3.17) [compile]
v module-office : 0.0.1-SNAPSHOT [compile]
  v poi-ooxml : 3.17 [compile]
      poi : 3.17 (omitted for conflict with 4.0.0) [compile]
      poi-ooxml-schemas : 3.17 [compile]
```

(b) Tracing issue to dependency heirarchy

Figure 5.13: Tracing the issue reported in DocBleach (issue #1)

5.6 Related Work

In this section, we present other works that are closely related to our research. We divide the prior work into three main related areas: API usage and identifying the impact of API breaking changes in client applications.

5.6.1 API Usage

Lammel et al. [112] conducted a large-scale API usage analysis on projects hosted in the SourceForge Repository. They observed that less than half of the APIs are used by client programs. Wu et al. [108] extended the work of [112] to identify API change types and their usage within client projects. Businge et al. [113] studied 512 third-party Eclipse plugins and their usages of Eclipse APIs. They observed that 44% of these plugins use internal Eclipse APIs and that API usages may have an important impact on upgrading such plugins. While our work is inspired by this existing research, our approach differs from the previous work by being based on a formal knowledge representation and the use of Semantic Web inference services to identify API usage at an inter-project and global scale.

5.6.2 Impact of API Breaking Changes

Change in software systems has been studied, measured, and modeled intensively over the years. For example, Cossette et al. have shown that Java libraries “frequently and seriously change over time” [114], [115]. Throughout a large body of research, all studied real-world systems evolved in unanticipated ways with rippling consequences across modules. Many approaches were proposed to support this activity and reduce their costs for client applications.

Dig and Johnson [116] define a catalog of breaking and non-breaking changes. They observed that refactoring accounts for 80% of the changes that break client systems. Raemaekers et al. [117] present four stability metrics based on method changes and removals. The authors investigate their metrics behavior by performing a historical analysis of stability and impact on 140 clients of the Apache Commons Library. Xavier et al. [107] conducted an extensive empirical study on 317 real-world Java libraries, 9K releases, and 260K client applications to investigate the impact of API breaking changes on client applications. They report that only 2.54% of the clients are potentially impacted, and larger and popular libraries have a higher frequency of breaking changes. Decan et al. [118] found that about 1 in every 20 updates to a CRAN package was a backward incompatible change, accounting for 41% of the errors in released packages that depended on them.

Complex and changing dependencies are a pain to work with for many developers [119] and have led to common expressions like “DLL hell” and “dependency hell.” In our work, we address some of these challenges, by extending the scope of the impact analysis for breaking changes, to include cross-project and even cross-ecosystem dependency analysis. Our approach is also able to detect binary incompatibilities introduced at different transitive levels of library dependencies, which may lead otherwise to unexpected runtime behavior.

5.7 Chapter Summary

The changing software engineering landscape with its global software development processes allows projects and organizations to take advantage of the plethora of features and functionality provided by existing third-party libraries. However, similar to other software, these external libraries also undergo changes which might introduce binary incompatibilities in client applications.

In this chapter, we presented an ontological modeling approach that describes a shared taxonomy of object-oriented programming and dependency management concepts. Our approach uses multi-layers of abstraction to provide a generic analysis approach and also supports the seamless integration of knowledge resources found in the software engineering domain. Given the expressiveness of our ontologies, we can take advantage of inference services provided by the Semantic Web, to infer additional knowledge that supports novel types of breaking change analysis. In this chapter, we discuss how our Semantic Web-enabled cross-project impact analysis service that can guide API producer in selecting an appropriate mitigation strategy. We also show, that by integrating different knowledge resource, API user can identify potential binary incompatibility errors that are usually not discovered during a program compilation and build.

In the next chapter, we present another application of SBSON – an API-level vulnerability impact analysis service.

Chapter 6

6 Recovering Semantic Traceability Links between APIs and Security Vulnerabilities

6.1 Introduction

According to a report in 2012 [120], OSS libraries and frameworks form 88% of the code in applications globally. In this context, development teams face a challenge in not only identifying and keeping track on which libraries a project and its third-party components depend on, but also which version of a particular library is being used by them. Libraries as any other software are susceptible to security vulnerabilities, which requires developers to fix or upgrade affected versions of a library in a timely fashion to mitigate potential security threats. As the study in [120] shows, 26% of these OSS frameworks/libraries suffer from vulnerabilities that often remain undiscovered. In 2017, “Using Components with Known Vulnerabilities” was ranked 9th in the OWASP Top Ten list of software security flaws [121], with some of the largest vulnerability breaches to date have been exploits of known vulnerabilities in components.

Several approaches have been introduced in the literature [122] to minimize the introduction and exploitation of software security vulnerabilities. These approaches fall in two main categories. The first category requires organizations to create barriers that prevent developers and end-users from performing potential risky actions, e.g., runtime protection. While this category can reduce the exposure to vulnerabilities, it does not address the fundamental cause of such vulnerabilities. The other category involves techniques that avoid or reduce the introduction of potential vulnerabilities already at the development stage, by introducing and applying best secure coding practices e.g., black-box testing, and static analysis. Unfortunately, most of these analysis techniques are limited to artifacts created within a project context and do not consider the reuse and sharing of third-party components across their own project boundaries in their analysis.

Different specialized Software Vulnerability Databases (SVDBs) (e.g. National Vulnerability Database (NVD)[30]) have been introduced by the Information Security domain to help track software vulnerabilities and their potential solutions. These SVDBs were introduced in response to the increasing number of software attacks, which are no longer limited to a project but often affect millions of computers and hundreds of different systems. These repositories can be considered as trusted information silos which are typically not directly linked to other software repositories, such as source code repositories containing reported instances of these problems.

In this chapter, we introduce a novel approach which establishes traceability links between security and software databases for automatically tracing source code vulnerabilities at the API level across project boundaries. More specifically, we integrate our software build system ontology (SBSON) with other our source code and versioning ontology (SEON) and a security vulnerability ontology (SEVONT). Based on the standardized knowledge representation (ontologies), we are now able to introduce new types of vulnerability analysis at a global scale.

6.1.1 Motivating Example

Existing research on recommending APIs to developers (e.g., [2]) has focused on recommending potentially useful APIs to developers to reduce development and testing time.

For example, in [2], the authors explicitly recommend developers to use an older version of Apache Derby (version 10.1.1.0) due to its widespread usage/popularity. However, like any other software project, Apache Derby is also susceptible to security vulnerabilities. By recommending this particular older version of Derby, the author in [2] actually recommended a version of Apache Derby which has two known security vulnerabilities (Table 6.1). These known vulnerabilities had already been published in the National Vulnerability Database (NVD) repository.

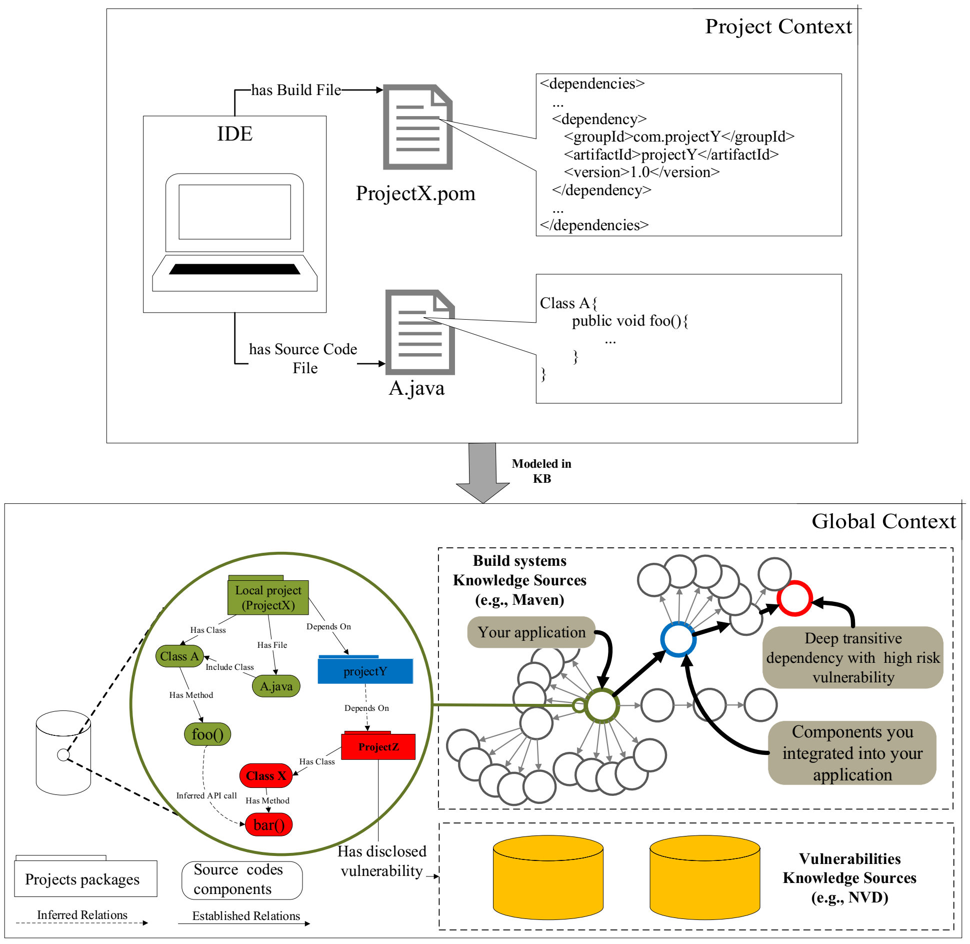
As shown in this example, the author of the paper was most likely unaware of these reported vulnerabilities, with one of the reasons being that this information is not readily available to developers. Making this information readily available to maintainers and security experts would

[30] https://nvd.nist.gov/

allow for seamless knowledge integration and sharing. Furthermore, by using standardized and formal knowledge representation techniques (e.g., Semantic Web and its technology stack), novel analysis approaches across knowledge boundaries at both the intra and inter-project level can be introduced.

Table 6.1: Example of Derby versions and their dependent projects in Maven

Derby version	Release Year	# vulnerabilities in NVD	Direct dependencies in Maven Repository
10.1.1.0	2005	3	382
10.5.3.0	2009	1	0
10.11.1.1	2014	0	36



```
<dependencies>
  ...
  <dependency>
    <groupId>com.projectY</groupId>
    <artifactId>projectY</artifactId>
    <version>1.0</version>
  </dependency>
  ...
</dependencies>
```

```
Class A{
      public void foo(){
            ...
      }
}
```

Figure 6.1: Integrating code and build information with knowledge from other originally heterogeneous resources

Figure 6.1 shows an illustrative example of an IDE with an open Maven POM (ProjectX.pom) and Java file (A.java). In our approach, we extend a developer's accessible knowledge from local project's pom and Java files, to knowledge resources outside the current project boundaries. Using our knowledge modeling approach, we can now integrate, share and reason upon these heterogeneous resources (even at a global scale). In this example, such a knowledge base includes project-specific resources (e.g., issue tracker, versioning repositories) as well as resources external to the project, such as NVD and Maven build dependencies from other projects. Using the reasoning services provided by the Semantic Web, we can now infer direct and indirect dependencies for the local project (ProjectX in Figure 6.1). In addition, giving the bi-directional links in our modeling approach, we can expand our analysis to answer questions like this: Which projects might be directly or indirectly affected by a vulnerable component/library? In our example, ProjectX has an indirect dependency on ProjectZ (via ProjectY's transitive dependencies) with ProjectZ being a vulnerable component.

As our example illustrates, integrating source code information with other knowledge resources (e.g., vulnerability and build repositories) can support new types of dependency and vulnerability analysis even at a cross-project boundary (global) scale. In addition, analysis results can now be made directly available in our knowledge model, to be reused by other analysis tools. For example, an existing APIs recommendation tool can now also consider in its recommendation if a direct/indirect recommended API may contain a vulnerability. Another example would be an automatic notification of developers when an already used API becomes exposed to a potential vulnerability.

Note: Earlier versions of this work are published in the *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, 2017 [123], the *IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Ottawa, 2016 [124], and *Science of Computer Programming*, Volume 121, 2016 [125].

6.2 Background

6.2.1 Security Vulnerability Databases

In the software security domain, a software vulnerability refers to mistakes or facts about the security of software, networks, computers or servers. Such vulnerabilities represent security risks to be exploited by hackers to gain access to system information or capabilities [120]. As discussed in [126] new software vulnerabilities are often first reported in software repositories (e.g., issue trackers, mailing lists) of the affected projects or mentioned on Q&A sites (e.g., StackOverflow). A common characteristic of early vulnerability reporting is that information about vulnerabilities is dispersed across multiple resources and their descriptions tend to be incomplete, inconsistent and informal. Advisory databases (e.g., NVD) were introduced to address some of these shortcomings. Their objective is not only to provide a central place for reporting vulnerabilities, but also to standardize their reporting. The Common Vulnerabilities and Exposures (CVE)[31] dataset creates a publicly available dictionary for vulnerabilities, allowing for a more consistent and concise use of security terminology in the software domain. Once a new vulnerability is revealed and verified by security experts, this vulnerability and other relevant information (e.g., unique identifier, source URL, vendor URL, affected resources and related vulnerabilities from the same family group) will be added to the CVE database. The source URL refers to the vulnerability (e.g., application vendor, external security advisories) by linking directly to the commit that contains the source code for patching or a document that describes on how to patch the vulnerability. In addition to the CVE entry, each vulnerability will also be classified using the Common Weakness Enumeration (CWE)[32] database. The CWE, therefore, provides a common language to describe software security weaknesses, by classifying them based on their reported weaknesses. NVD, CVE, and CWE can all be considering being part of a global effort to manage the reporting and classification of known software vulnerabilities.

6.2.2 Vulnerability Detection Techniques

In the SE domain, vulnerability detection techniques (tools) provide project managers and developers with security vulnerability assessments and quantitative insight into the effectiveness of a projects' security controls. The traditional techniques used to audit software projects against security vulnerabilities are based on static analysis tools (e.g., FindBugs[33]) and vulnerability scanners (e.g., OWASP Dependency-Check[34]).

[31] https://cve.mitre.org/
[32] https://cwe.mitre.org/
[33] http://findbugs.sourceforge.net/
[34] https://www.owasp.org/index.php/OWASP_Dependency_Check

Vulnerability scanner tools play a different role than traditional static analysis tools by scanning the security vulnerabilities in a software project based on some predefined rules (maintained by security engineers). In addition, the vulnerability scanner usually identifies project dependencies and checks if there are any known vulnerabilities publicly disclosed in existing vulnerability databases (vendor vulnerability database, or third-party database such as NVD, SecurityFocus[35], etc.). These scanners help to validate the inventory of third-party libraries in a project. In what follows we give a detailed example of the OWASP Dependency-Check vulnerability scanner tool which we used to evaluate our proposed approach discussed in this chapter.

OWASP Dependency-Check is a vulnerability scanner that analyzes the dependency definitions within a project's build file (e.g., a pom.xml file for Maven projects) and collects a set of coordinates called Evidence. There are three types of evidence collected: vendor, product, and version. The evidence for each build/dependency manager may vary from one to another. For example, the coordinates (evidences) for Java (Maven) are groupId, artifactId, and version. Node.js (NPM), Python (PyPi), and Ruby (Ruby Gems) use library name and version as their dependency coordinates. Dependency-Check matches these evidences with public data in NVD to identify and report the vulnerable libraries to the user.

6.2.3 The SEcurity Vulnerability ONTology (SEVONT)

Although several SVDBs have been introduced to address the identification and management of software vulnerabilities, software developers fail to take full advantage of these SVDBs due to vast amount of security vulnerability data available. The situation is further complicated by the fact that these heterogenous SVDBs introduce ambiguity in their datasets, resulting in diverse data modeling results. To address these issues, Alqahtani [127] introduced SEVONT (Figure 6.2), an abstraction hierarchy of software security vulnerabilities analysis ontologies, which reconciles and integrates heterogeneous vulnerability data from several SVDBs. The SEVONT ontology includes the following important domain concepts:

[35] http://www.securityfocus.com/bid

- *Vulnerability*. In software security, a vulnerability refers to a flaw in the system which is introduced by reusing vulnerable (external) software components or inadvertent coding mistakes by developers (e.g., bad coding practices).
- *Product*. Software products are assets of the organization which are a result of a software development process (e.g., hardware, artifacts).
- *Attacker*. Attackers, either internal or external entities of the system, attack a product to perform malicious actions which attempt to break the security of a software system or its components.
- *Attack*. Attacks are malicious actions designed to compromise the security of a system. Security experts analyze these attacks to study the behavior of attackers, estimate the cost of attacks, and determine their impact on overall system security.
- *Countermeasure*. A countermeasure is a mechanism used to protect a system from potential vulnerability attacks (e.g., patch development, encryption/decryption enhancement, and updated system security configurations).

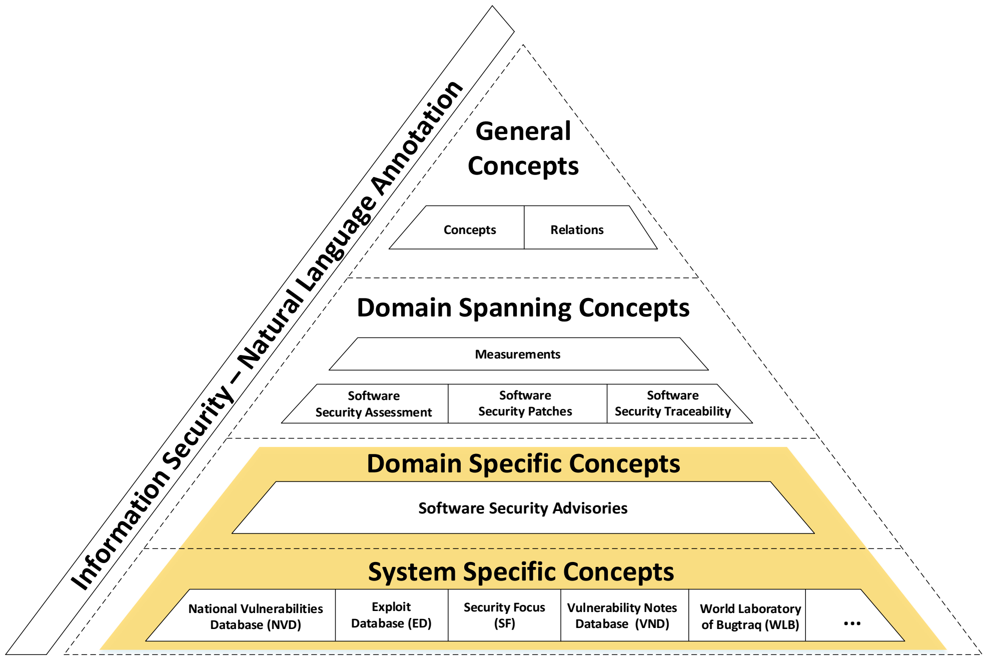

Information Security – Natural Language Annotation
General Concepts
Concepts
Relations
Domain Spanning Concepts
Measurements
Software Security Assessment
Software Security Patches
Software Security Traceability
Domain Specific Concepts
Software Security Advisories
System Specific Concepts
National Vulnerabilities Database (NVD)
Exploit Database (ED)
Security Focus (SF)
Vulnerability Notes Database (VND)
World Laboratory of Bugtraq (WLB)
...

Figure 6.2: Overview of the SEVONT ontologies

6.3 SV-AF: Security Vulnerability Analysis Framework

It is generally accepted that inadvertent programming mistakes can lead to software security vulnerabilities and attacks [120]. Mitigating these vulnerabilities can become a major challenge for developers, since not only their own source code might contain exploitable code, but also the code of third-party APIs or external components used by their system. In what follows we introduce a methodology to guide developers in identifying the potential impact of vulnerabilities at both the system and global level. Our methodology integrates knowledge from the (1) SBSON, (2) SEON and (3) SEVONT ontologies (see Figure 6.3). In what follows, we detail the approach used to align these ontologies and the reasoning capabilities provided by this unified knowledge representation. It should be noted that in order to improve readability, we use prefixes as substitutes to the fully qualified names of our ontologies. The ontology prefixes used in this chapter can be dereferenced using the URIs shown in Appendix A.

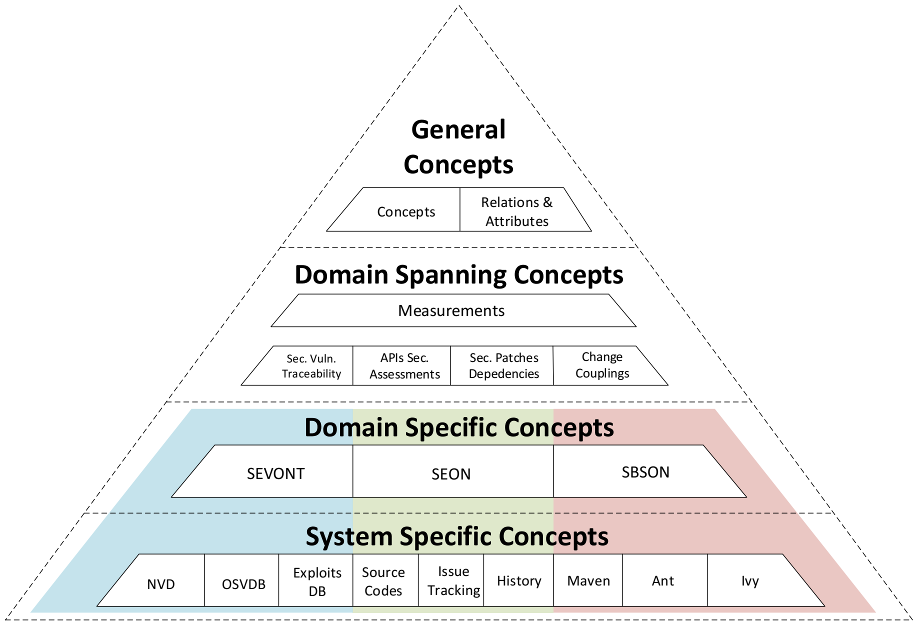


Figure 6.3: Overview of the integrated SBSON, SEON, and SEVONT ontologies

6.3.1 Ontology Alignment

For us to take full advantage of the knowledge captured in the SBSON, SEON and SEVONT ontologies, we apply ontology alignment techniques to establish traceability links among these ontologies. This linking process requires either shared concepts across knowledge resources or identifying semantically identical or similar concepts within the different knowledge sources. These links reduce the semantic gap between these ontologies and are essential pre-requisites for supporting seamless knowledge integration.

6.3.1.1 Alignment of the SBSON and SEVONT ontologies

In our work, ontologies undergoing an alignment are treated as uncertain graphs. In an uncertain graph, edges are associated with an uncertainty value which measures the strength of connectivity between nodes and/or edges [128]. An uncertain directed graph is defined as $G = (V, E, \omega)$, where V is a set of nodes, E is a set of edges (x, y), and ω: E → [0, 1] is the weight assignment function (e.g., ω(x, y) = 0.3 means the associated value on edge (x, y) is 0.3). Uncertainty values are interpreted as probabilities.

In our model, V represents the modeled projects nodes from SBSON and SEVONT, E represents <owl:sameAs> relations (edges) between project instances, and ω: E → [0,1] is the weight assignment function. For example, in Figure 6.4, the project instance $V_m$from SBSON graph is similar to vulnerable product instance V_n from SEVONT graph through <owl:sameAs> (ω(e)) edge.

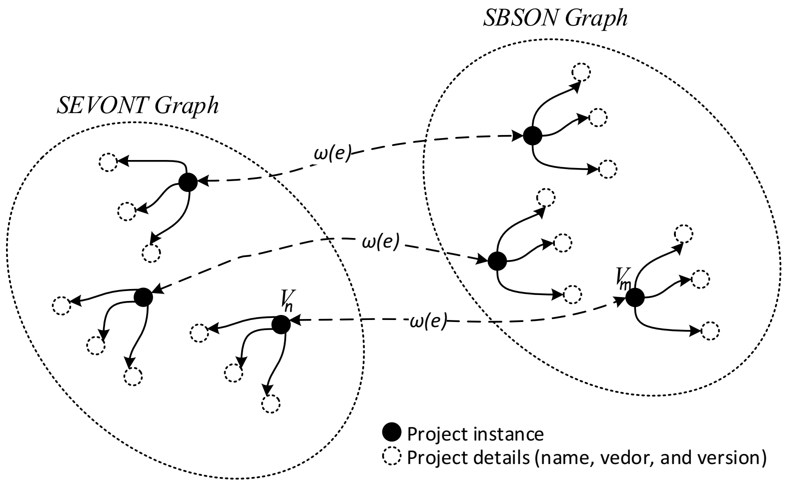


Figure 6.4: SV-AF knowledge base similarity graphs

We use the Probabilistic Soft Logic (PSL) framework [129] to establishes weighted links between the SBSON and SEVONT ontologies. PSL uses continuous variables to represent truth values instead of the traditionally used Boolean values [129]. The resulting probability distribution is captured in a graph model, which can then be reasoned upon.

For example, in Figure 6.5, the rule states that two instances A and B with similar names defined in different source ontologies are likely to be similar. “similarID” is a similarity function implemented using the Levenshtein similarity metric. Rules in PSL are labeled with non-negative weights. For example, the weights of the rule in Figure 6.6 indicate that projects with the same name and version are more likely to be similar than projects with same names only. Using PSL, we establish <owl:sameAs> relations between similar instances found in the two ontologies. The number of possible instance pairs for these two ontologies is $|SBSON| \times |SEVONT|$.

```
type(A, instance) ∧
type(B, instance) ∧
name(A,X) ∧
name(B,Y) ∧
similarID(X,Y) ∧
A.source ≠ B.source
→ similar(A,B)  weight:0.5
```

Figure 6.5: PSL rule identifying similar projects with the same name

```
type(A, instance) ∧
type(B, instance) ∧
name(A,X) ∧
name(B,Y) ∧
similarID(X,Y) ∧
A.source ≠ B.source
version(A,Z) ∧
version(B,K) ∧
similarID(Z,K)
→ similar(A,B)  weight:0.8
```

Figure 6.6: PSL rule identifying similar projects with the same name and version

In this example, similarity among instance pairs is determined based on the extracted literal information such as name, version and vendor. We used the PSL framework classifier to compute the similarity weights for the <owl:sameAs> links. For training purposes, we created a dataset with manually labeled instance links to train the PSL classifier to establish the weights for our pre-defined rules. Derived similarity weights for each instance pair (see Figure 6.7) are captured by the domain-spanning <measure:SimilarityMeasure> concept. Given the weighted alignment links between the two ontologies, a SPARQL query can now be written, to retrieve the vulnerability information from the SEVONT ontology and their corresponding instances in SBSON ontology based on a given similarity threshold.

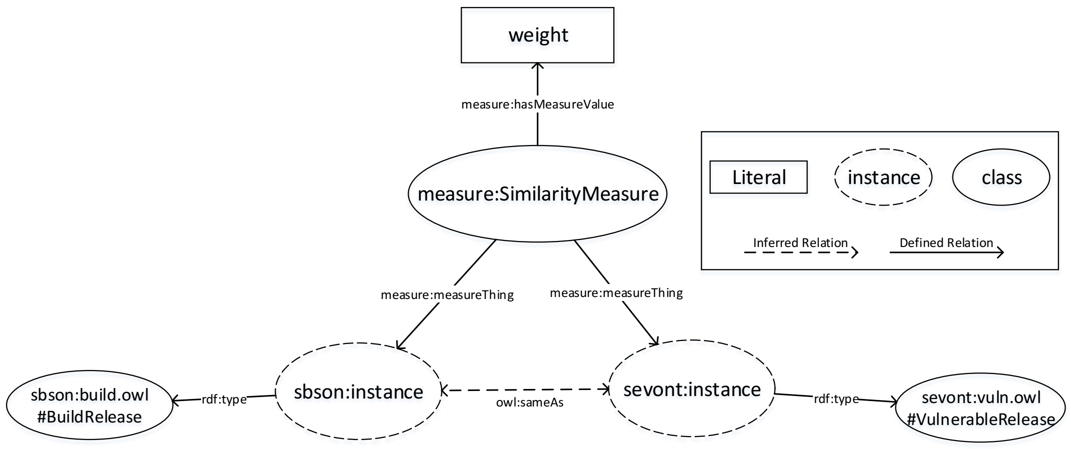


Figure 6.7: SV-AF’s weighted similarity modeling

6.3.1.2 Alignment of the SEON and SEVONT ontologies

Disclosed vulnerabilities often contain references to patch information, such as explicit revisions/commits in which the vulnerability has been fixed. Having this information available, we can perform terminology matching to align instances from both data sources. For the alignment process, we take advantage of reasoning services provided by the Semantic Web to infer implicit relationships between vulnerabilities and commits. More specifically, for the alignment, we take advantage of SWRL rules to establish links between vulnerability and commit instances. This alignment takes place if any of the two semantic rules are satisfied:

Rule 1: Vulnerability ID is explicitly mentioned in a commit message (see Figure 6.8).

Rule 2: Commit/revision ID is explicitly mentioned in the patch reference of a vulnerability (see Figure 6.9).

```
Commit(?c),
fixNVDIssue(?c,?ID),
Vulnerability(?v),
hasVulnerabilityID(?v,?ID)
→ vulnerabilityFixedIn(?v,?c)
```

Figure 6.8: SWRL rules for aligning SEON and SEVONT when a commit message contains a vulnerability reference

```
Vulnerability(?v),
hasPatch(?v,?p),
hasFixRevision(?p,?ID),
Commit(?c),
hasCommitID(?c,?ID)
→ vulnerabilityFixedIn(?v,?c) )
```

Figure 6.9: SWRL rules for aligning SEON and SEVONT when a vulnerability patch contains a commit reference

6.3.1.3 Overview of the integrated ontologies in SV-AF

The result of this alignment processes is SV-AF, a unified representation that supports impact analysis of known vulnerabilities across heterogeneous software repositories. SV-AF provides a seamless integration of build dependency, source code, versioning history, and software vulnerability concepts and relations across different abstraction layers. The OWL classes and object properties used for our API-level vulnerability impact analysis are shown in Figure 6.10 (data properties have been again omitted to improve readability of the figure).

The core concepts used for our vulnerability analysis are <sevont:VulnerableRelease>, <sbson:BuildRelease>, and <sevont:SecurityPatche>. A <sevont:VulnerableRelease> is a software <main:Release> within the NVD database with a known <sevont:Vulnerability>. A <sbson:BuildRelease> is a software release within the Maven ecosystem. Using our ontology alignment process, we infer that a given <sevont:VulnerableRelease> is <owl:sameAs> a specific <sbson:BuildRelease>. As such, the <sevont:VulnerableRelease> inherits the properties of the original <sbson:BuildRelease>, for example, the <sevont:VulnerableRelease> now <main:dependsOn> other <sbson:BuildRelease>. Given the support for bi-directional links in our model, a project release can now be identified as being potentially affected by a vulnerability when it directly or indirectly reuses a vulnerable release.

Whenever a <main:Project> is identified to be affected by a vulnerability, a <version:Committer> commits a new version of a <version:VersionedFile> containing a <sevont:SecurityPatch> through a version system (e.g., SVN). Versioned files are <main:File> managed by a version control system. A <sevont:SecurityPatch> corresponds to code changes introduced to fix some existing <sevont:VulnerableCode>, which is part of a <code:CodeEntity>, such as <code:ComplexType>(i.e., a Class, Interface, Enum, etc.) or a <code:Method>. For example, if a class or method is modified during a security patch, then this code change can be used to locate the original <sevont:VulnerableCode> individual. The OWL classes, <sevont:SecurityPatch> and <sevont:VulnerableCode>, are linked in our model through the object property identifies.

6.3.2 Knowledge Inferencing and Reasoning

In addition to transitivity and subsumption inferencing provided by our SBSON and SEON ontologies (see Section 5.4.2), SV-AF also takes advantage of the inbuilt <owl:sameAs> inferencing services to trace APIs and their vulnerabilities across knowledge boundaries.

The <owl:sameAs> predicate is used to align two concepts from different ontologies. For the SBSON-SEVONT alignment process, we create weighted alignment links between the two ontologies. These weighted (based on the similarity threshold) links, are used to infer the <owl:sameAs> predicate between instances within the ontologies. Simple queries such as the SPARQL query in Figure 6.11 can now be executed to take advantage of the <owl:sameAs> predicate (if inference is enabled) and retrieve facts across two or more ontologies. Without inferencing, the query result set would be empty since SBSON has no triple with any knowledge of vulnerabilities (line 5 of Figure 6.11). However, with inference enabled (line 1 of Figure 6.11), vulnerabilities for releases in SBSON can be identified based on the established <owl:sameAs> instances in the two ontologies.

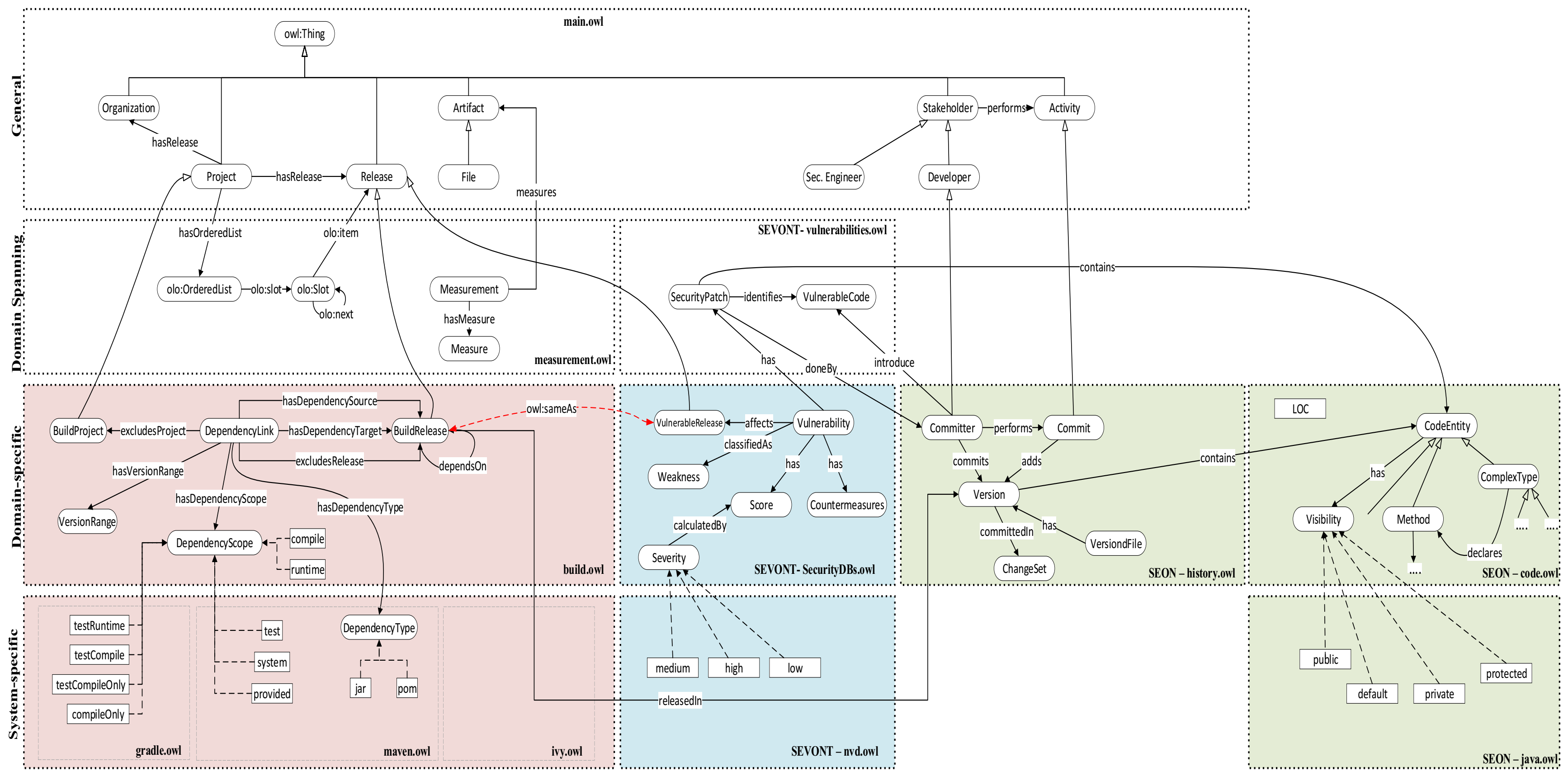


Figure 6.10: The SV-AF ontology concepts involved in API-level vulnerability impact analysis

```
define input:same-as "yes"
SELECT ?vulnerability ?release
WHERE {
  ?release rdf:type sbson:BuildRelease.
  ?release sevont:hasVulnerability ?vulnerability.
}
```

Figure 6.11: SPARQL query returning vulnerable projects based on the owl:sameAs inference

6.4 Case Studies

In what follows, we introduce three case studies which we conducted to illustrate the applicability of our knowledge modeling approach. For the first case study, we identify project releases in the Maven Central repository which contain known security vulnerabilities disclosed in the NVD database. The objective of this case study is to evaluate the applicability of our alignment process. For the second case study, we illustrate how our semantic rules can identify explicit and implicit security vulnerabilities by inferring transitive dependencies across SBSON and SEVONT. Finally, the third case study demonstrate the applicability of our modeling approach in analyzing API-level security vulnerability impacts across software components.

6.4.1 Case Study Data

For the data collection and extraction in our case studies, we rely on two data sources: the NVD database and the Maven Central repository. We download the latest version of the Maven repository (Table 6.2) and all NVD vulnerability xml feeds from 1990 to 2016 (Table 6.3). For case study #1, we used the releases and unique vulnerable products to evaluate the alignment of the SBSON and SEVONT ontologies.

Table 6.2: Maven Repository statistics

Repository	Projects	Releases	Last Update
Maven Central	130,895	1,219,731	2016-01-28 16:34:07 UTC

Table 6.3: NVD database statistics

Repository	# unique vulnerabilities	# unique vulnerable products	Period
NVD [130]	74945	109212	1990 - 2016

For case study #2, the objective was to identify the potential transitive impact set of some vulnerable components on other systems. For this study, we selected five Apache projects (Table 6.4) which are using the Maven repository for its build management. The main selection criteria for these projects was that at least some of their releases contain known vulnerabilities (which we had identified in case study #1) and the projects are commonly reused by other projects. These five subject systems vary in size (classes and methods) and application domain. Wss4J[36] is a Java implementation of the primary security standards for Web Services, Httpclient[37] is responsible of provides reusable components for client-side authentication, HTTP state management, and HTTP connection management. Apache Derby[38] is an open source relational database implemented entirely in Java, Hibernate Validator[39] allows expressing and validating application constraints using annotation-based constraints, and Apache OpenJPA[40] is a Java persistence project that can be used as a stand-alone plain old Java object (POJO) persistence layer or be integrated into any Java EE compliant container.

Table 6.4: Subject systems and sizes for transitive dependencies analysis

ID	Subject Systems	Version	Size	
			Classes	Methods
P1	Wss4J	1.6.16	167	1610
P2	Httpclient	4.1	209	1180
P3	Derby	10.1.1.0	967	16354
P4	Hibernate-validator	4.1.0.Final	325	2642
P5	Openjpa	1.1.0	1201	18640

6.4.2 Case Study 1: Identifying vulnerable projects in Maven Repository

Objective: The goal of this study is to evaluate the performance of our semantic similarity linking approach used to align two domain specific ontologies.

[36] https://ws.apache.org/wss4j/
[37] https://hc.apache.org/httpcomponents-client-ga/
[38] https://db.apache.org/derby/
[39] http://hibernate.org/validator/
[40] http://openjpa.apache.org/

Approach: In order to link these two ontologies (SEVONT and SBSON), we use the PSL framework to align project specific information found in both ontologies. We trained PSL using a corpus of 524 randomly selected project instance pairs and their manually derived similarity links. Next, we executed our PSL alignment rules for this training dataset to train our approach. Based on this training, two concept instances, A and B, located in different ontologies (¬*SameSource*) can now be aligned (with a degree of certainty), if both have the same names, similar vendors, and same version numbers (PSL rule in Figure 6.12). The SameName, SimilarVendor, and SameVersion similarity functions are implemented using the Levenshtein distance metric. In this rule, the SameProject(A,B) is given a weight of 0.9 based on results from the PSL training set. Figure 6.13 shows the PSL inference results for our training dataset, using different weights (ranging from a minimum of 0.04 to a maximum of 0.42) for the SameProject(A,B) alignment.

Using the semantic rule from Figure 6.12, PSL can now perform maximum a posteriori (MPE) reasoning [129] to infer the most likely values for a set of propositions and observed values for the remaining (evidence) propositions. For a full discussion on MPE reasoning, we refer the reader to [129]. The results of the PSL inference is a set of $A \times B$ SameProject weights with a [0..1] range; 0 corresponds to two concept instances with no similarity and 1 corresponds to an exact (100%) match.

```
Source(A,SnA) ∧
Source(B,SnB) ∧
¬SameSource(SnA,SnB) ∧
Name(A,X1) ∧
Name(B,Y1) ∧
SameName(X1,Y1) ∧
Vendor(A,X2) ∧
Vendor(B,Y2) ∧
SimilarVendor(X2,Y2) ∧
Version(A,X3) ∧
Version(B,Y3) ∧
SameVersion(X3,Y3)
⇒ SameProject(A,B)  weight:0.9
```

Figure 6.12: PSL SameProject Rules

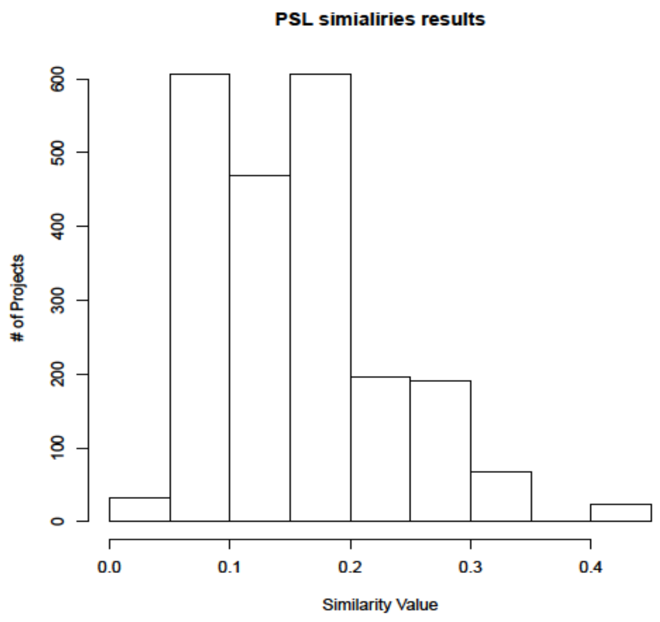


Figure 6.13:PSL SBSON-SEVONT similarity inference results

As part of our knowledge modeling approach, we materialized the inferred semantic instance links (<owl:sameAs>) between the SEVONT and SBSON ontology, making this inferred knowledge a persistent part of our knowledge model. As part of this materialization process, we also add weights for each link, which are the inferred similarity values using the domain spanning similarity measure (<measure:SimilarityMeasure>) from our unified knowledge model.

Findings. Our study showed that 0.062% of all Maven projects contain known security vulnerabilities that have been reported in the NVD database. An example for such a vulnerability is shown in Table 6.5.

Table 6.5: Example of a linked SBSON-SEVONT vulnerability

SBSON Fact	SEVONT Fact	Corresponding Vulnerability
org.sonatype.nexus:nexus:2.3.1	sonatype:nexus:2.3.1	CVE-2014-0792

Further analysis of our results showed that projects might often suffer not only from one but from **multiple** vulnerabilities. We found that 48.8% of the 750 identified vulnerable project releases suffer from multiple security vulnerabilities, with PostgreSQL 7.4.1 being the most vulnerable project in the dataset, containing 25 known vulnerabilities. Having this additional insight can guide developers in their system update and upgrade decisions by avoiding the reuse of APIs/components with known security vulnerabilities or components that might be prone to vulnerabilities.

For example, in December 2010, Google released its Nexus S smartphone[41]. The phone was originally running on Android 2.3.3 – an Android version that already was exposed to the security vulnerability discussed in Table V. While the Nexus S received regular Android OS updates up to Android Version 4.2, an actual fix of the reported vulnerability (CVE-2013-4787) was only introduced with Android 4.2.2. However, this new Android version was no longer supported and distributed for the Nexus S, leaving existing users of the phone susceptible to attacks. Our analysis showed that the same vulnerability can affect multiple releases of a product. For example, security vulnerability CVE-2013-4787[42] has been reported for five different Android versions (Table 6.6). This information can help product maintainers to ensure consistent patching and regression testing across product lines or different product versions.

Table 6.6: Critical Vulnerabilities for Android Project

Android Version	CVE-IDs	# of direct dependencies
com.google.android:android:2.2.1	CVE-2013-4787	360
com.google.android:android:2.3.1	CVE-2013-4787	176
com.google.android:android:2.3.3	CVE-2013-4787	351
com.google.android:android:3.0	CVE-2013-4787	34
com.google.android:android:4.2	CVE-2013-4787	1

Evaluation: In what follows, we evaluate the accuracy of aligning project instances (<owl:sameAs>) between our SBSON and SEVONT ontologies. During the first step of this evaluation, we compared using precision, recall and F1 measure, the impact of different similarity weight thresholds ($w = 0.1$, $w = 0.2$, $w = 0.3$, and $w = 0.4$) on the inferred links created by the PSL alignment process. Precision is calculated, with true positives being the number of project instance pairs correctly classified as similar, while false positives corresponds to the number of non-similar instance pairs that are incorrectly classified as same projects. For

[41] https://en.wikipedia.org/wiki/Nexus_S
[42] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4787

Recall, false negatives correspond to the number of non-similar instance pairs that are incorrectly classified as being similar projects. The F1-score is the harmonic mean of precision and recall, giving equal weight to both measures.

Our analysis (Table 6.7) showed that an increase in the similarity threshold from 0.1 (low similarity) to 0.4 ((higher similarity) only had a limited effect on the precision (decrease from 0.98 to 0.75) while recall significantly dropped - down from 0.68 to 0.01.

Table 6.7: Weighted owl:sameAs link evaluation

	Precision				
Data Size	**w=0.0**	**w=0.1**	**w=0.2**	**w=0.3**	**w=0.4**
500	0.77	0.88	0.98	0.93	0.75
	Recall				
	0.77	0.68	0.30	0.03	0.01
	F1-score				
	0.77	0.77	0.46	0.05	0.01

A further manual inspection of the inferred links showed that the low recall for the higher threshold values is due to the inconsistent capturing of vendor information within the two ontologies. NVD relies on the common name to identify a vendor, whereas Maven uses the fully qualified package name as the vendor name. For example, using w=0.0, org.apache.cxf:cxf:3.0.1, org.apache.geronim.configs:cxf:3.0.1 and org.apache.geronimo.plugins:cxf:3.0.1 in SBSON will be considered the same instance as apache:cxf:3.0.1 in SEVONT and therefore correctly linked. However, using a higher similarity threshold, these instances will no longer be linked. We use the similarity weight of $w = 0.1$ in all subsequent experiments due to its high F1-score.

6.4.3 Case Study 2: Identifying open source components that are directly and indirectly dependent on vulnerable components.

Objective: In this study we evaluate how our framework can support the analysis of potential security vulnerability impacts on dependent software components.

Approach: For this case study, we extend our analysis to include transitive closure dependencies (Figure 6.14) that not only identifies components that are directly but also indirectly affected by known vulnerabilities. For this impact analysis, we selected 5 open source

Java projects (Table 6.4) with known security vulnerabilities. In this case study, we do not distinguish if a component uses (calls) a vulnerable component or not.

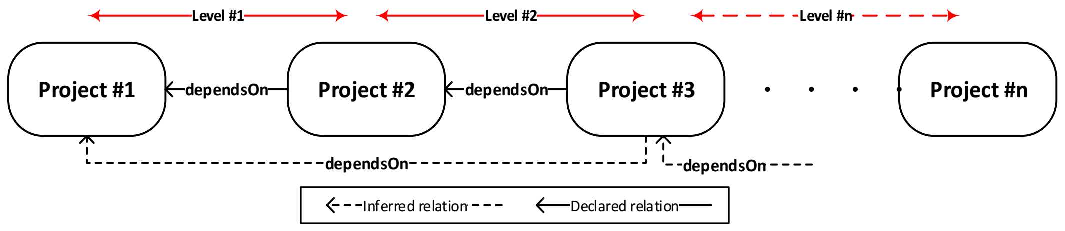


Figure 6.14: Inferred project dependencies in SBSON

Findings: We now report on results from our transitive dependency analysis, which also highlights the benefits of our knowledge modeling approach, its ability to integrate knowledge resources while taking advantage of inference services provided by the SW. Given the bi-directional links in our model between the NVD and the Maven repository, our analysis is no longer limited to identifying only direct dependencies on vulnerable components. Instead, given a vulnerable component, we can now also provide a more holistic analysis, which can identify all projects which directly and indirectly depend on a given vulnerable component.

Table 6.8 provides a summary of our analysis. It should be noted, that we limited the scope of our transitive analysis to only three levels of transitivity, in order to restrict the result set. For example, the vulnerable project Hibernate-validator 4.1.0 (P4) has a potential impact set of 3805 direct dependent projects (level 1) and 128109 dependent projects when we consider an additional two levels of transitivity (level 3).

Table 6.8: Summary of transitive dependencies on vulnerable components

ID	**Component Name**	**# Vulnerabilities**	**CVE-IDs**	**# Transitive Dependencies**		
				Level 1	**Level 2**	**Level 3**
P1	Wss4j 1.6.16	2	CVE-2015-0227 CVE-2014-3623	336	639	73
P2	Httpclient 4.1	2	CVE-2011-1498 CVE-2014-3577	685	4961	41326
P3	Derby 10.1.1.0	3	CVE-2005-4849 CVE-2006-7216 CVE-2006-7217	385	37999	66147
P4	Hibernate-validator 4.1.0.Final	1	CVE-2014-3558	3805	39295	128109
P5	Openjpa 1.1.0	1	CVE-2013-1768	74	49460	141303

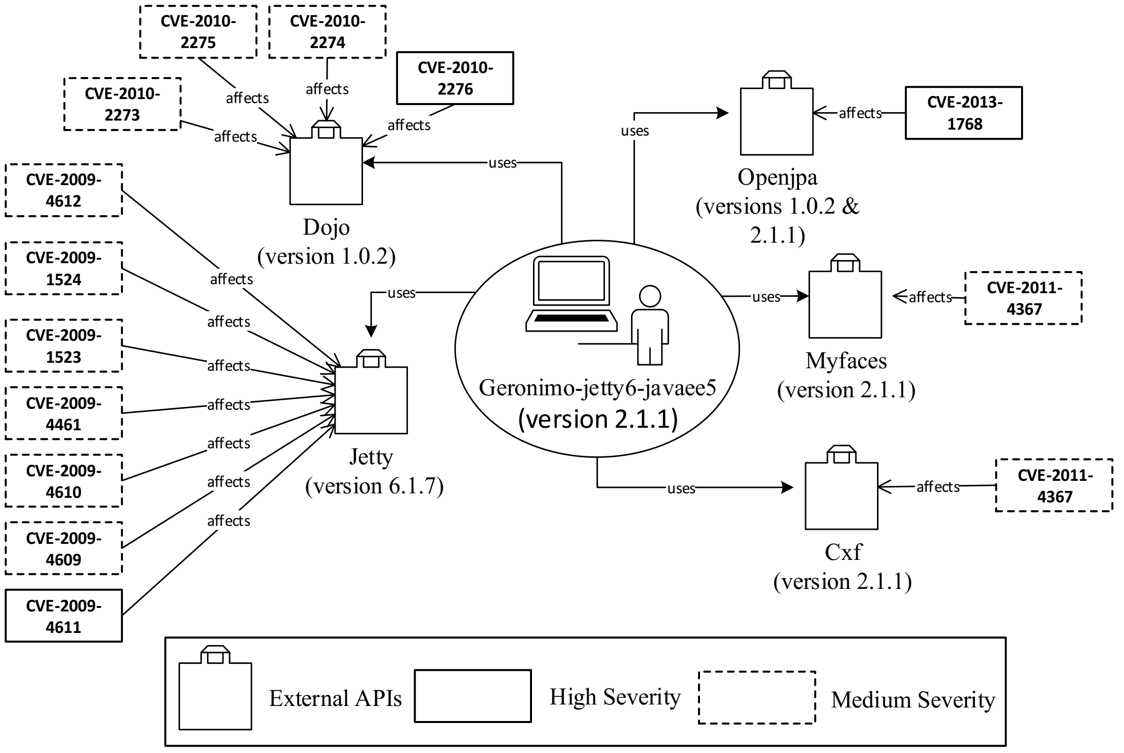


Figure 6.15: Geronimo-jetty6-javaee5 using 5 vulnerable projects (level 1 dependencies)

Figure 6.15 illustrates a typical usage scenario for modeling approach. While the Geronimo-jetty6-javaee5 (version 2.1.1) itself has no known vulnerability reported, the project depends on several components (level 1 dependencies) with known security issues (5 Java projects with a total of 15 known security vulnerabilities), making also Geronimo-jetty6-javaee5 potentially a very vulnerable component.

6.4.4 Case Study 3: API-level vulnerability impact analysis for CVE-2015-0227

Objective: The objective of our third case study is to show how our modeling approach can support the analysis and tracing of potential security vulnerability impacts at the API level of software components. Furthermore, the study also highlights again the flexibility of our modeling approach, in terms of its seamless knowledge and analysis result integration, as well as the use of Semantic Web reasoning to infer new knowledge.

Approach: For the case study, we take advantage of the same-as and transitive inference services provided by SV-AF to identify projects that are directly and indirectly affected by known security vulnerabilities. In addition, we also take advantage of transitive and subsumption inferences applied at the source code level to identify vulnerable APIs and trace their impact to external dependencies.

Case study setting: We use a publicly disclosed vulnerability, which has been reported in the NVD repository as CVE-2015-0227[43] and describes the following vulnerability for Apache WSS4J: "*Apache WSS4J before 1.6.17 and 2.x before 2.0.2 allows remote attackers to bypass the requireSignedEncryptedDataElements configuration via a vectors related to 'wrapping attacks'*." This vulnerability affects the management of permissions, privileges and other security features that are used to perform access control to Apache WSS4J versions before 1.6.17 and to version 2.x before 2.0.2.

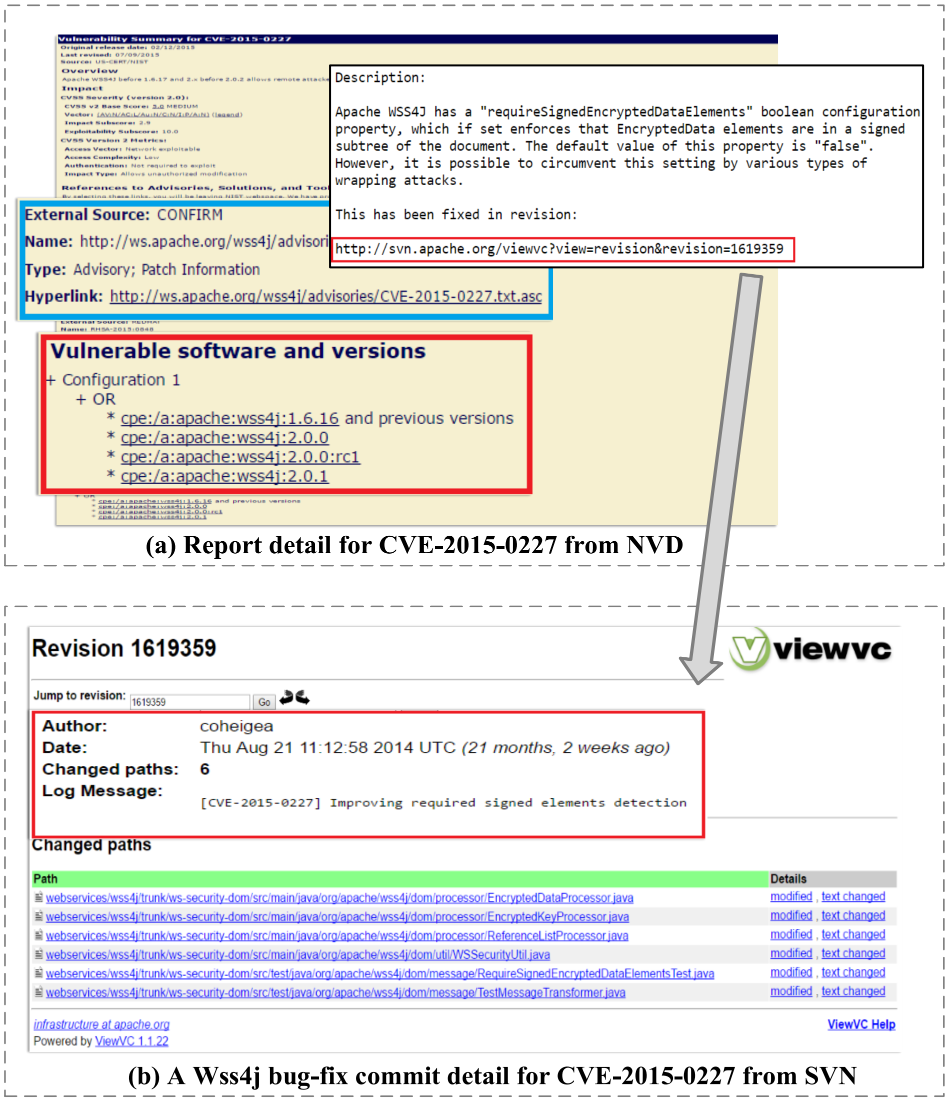
Apache WSS4J is an API which provides a Java implementation of the primary security standards for Web Services and is commonly used by projects as an external component. In this example, developers using any of the affected Apache WSS4J releases in their project must determine if their application is affected by this vulnerability. Existing source code analysis tools can identify whether the vulnerable code fragment (e.g., code fragment or variable) which has been reported in the NVD vulnerability is used directly within a project. However, they are not capable of identifying whether the external libraries used in the developer's project might have been affected by this vulnerability.

We now discuss how our approach takes advantages of the integrated knowledge from originally heterogeneous knowledge resources such as NVD, VCS (for only Apache WSS4J), and Maven to determine direct and indirect dependencies of vulnerable components. For this analysis, we extract and populate facts from a) NVD: information for the CVE-2015-0227 vulnerability (including patch references); b.) VCS: source code and commit messages for Apache WSS4J (version 1.6.16 and 1.6.17) and c.) Maven repository: all build dependencies on Apache WSS4J 1.6.16 (242 dependencies).

[43] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0227

Tracing vulnerability patch information to commit. Security databases provide descriptions of vulnerabilities, their potential effects, and corresponding patches (if applicable). The objective of our study is to establish a traceability link between the unique vulnerability identifier (CVE) and the commit which fixes this vulnerability. For establishing these links, we apply a two-step process, by first mining the NVD repository for patch links that include a reference to an entry in a versioning repository. We then extract all commit logs within the versioning repository that have a reference to a CVE-ID. Figure 6.16 shows an example of such a commit log message entry: "[CVE-2015-0227] Improving required signed elements detection."

Identifying vulnerable code fragments in APIs. A vulnerable code fragment corresponds to a set of lines of code (LoC), which has been modified to fix a vulnerability [131]. In our approach, we use the standard diff command to identify the vulnerable code fragments, by comparing it with its unpatched version. Figure 6.17 shows an excerpt of the diff output for WSSecurityUtil.java revisions r1619358 and r1619359. We further identify that method verifySignedElement contains the vulnerable code fragment. Using the same approach, we can now populate any method or class that has been either deleted or modified as part of a vulnerability fix (commit) in our sevont:VulnerableCode concept.



Vulnerability Summary for CVE-2015-0227
Original release date: 02/12/2015
Last revised: 07/09/2015
Source: US-CERT/NIST
Overview
Apache WSS4J before 1.6.17 and 2.x before 2.0.2 allows remote attacke
Impact
CVSS Severity (version 2.0):
CVSS v2 Base Score: 5.0 MEDIUM
Vector: (AV:N/AC:L/Au:N/C:N/I:P/A:N) (legend)
Impact Subscore: 2.9
Exploitability Subscore: 10.0
CVSS Version 2 Metrics:
Access Vector: Network exploitable
Access Complexity: Low
Authentication: Not required to exploit
Impact Type: Allows unauthorized modification
References to Advisories, Solutions, and Tool

External Source: CONFIRM
Name: http://ws.apache.org/wss4j/advisori
Type: Advisory; Patch Information
Hyperlink: http://ws.apache.org/wss4j/advisories/CVE-2015-0227.txt.asc

```
Description:

Apache WSS4J has a "requireSignedEncryptedDataElements" boolean configuration
property, which if set enforces that EncryptedData elements are in a signed
subtree of the document. The default value of this property is "false".
However, it is possible to circumvent this setting by various types of
wrapping attacks.

This has been fixed in revision:

http://svn.apache.org/viewvc?view=revision&revision=1619359
```

Vulnerable software and versions
+ Configuration 1
  + OR
    * cpe:/a:apache:wss4j:1.6.16 and previous versions
    * cpe:/a:apache:wss4j:2.0.0
    * cpe:/a:apache:wss4j:2.0.0:rc1
    * cpe:/a:apache:wss4j:2.0.1

(a) Report detail for CVE-2015-0227 from NVD

Revision 1619359

Jump to revision: 1619359 Go

Author:	coheigea
Date:	Thu Aug 21 11:12:58 2014 UTC *(21 months, 2 weeks ago)*
Changed paths:	**6**
Log Message:	`[CVE-2015-0227] Improving required signed elements detection`

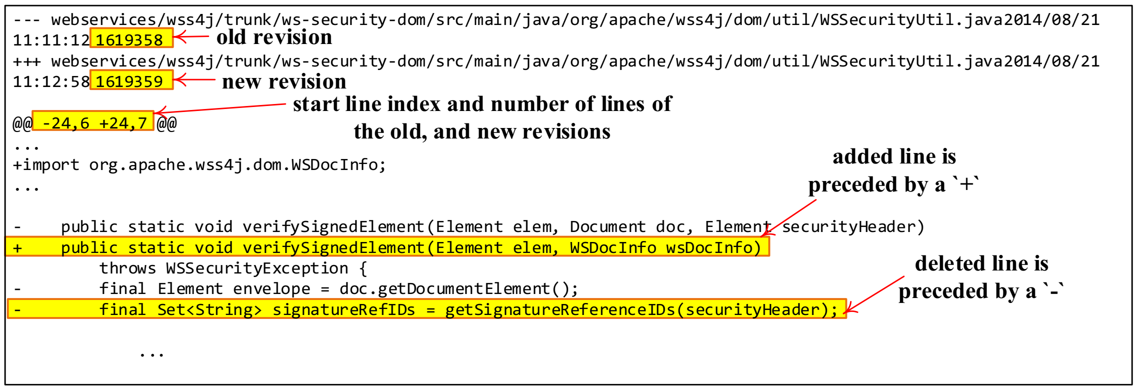
Changed paths

Path	Details
webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/processor/EncryptedDataProcessor.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/processor/EncryptedKeyProcessor.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/processor/ReferenceListProcessor.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/util/WSSecurityUtil.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/test/java/org/apache/wss4j/dom/message/RequireSignedEncryptedDataElementsTest.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/test/java/org/apache/wss4j/dom/message/TestMessageTransformer.java	modified , text changed

infrastructure at apache.org ViewVC Help
Powered by ViewVC 1.1.22

(b) A Wss4j bug-fix commit detail for CVE-2015-0227 from SVN

Figure 6.16: Extracting patch relevant information from NVD and commit messages



```
--- webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/util/WSSecurityUtil.java2014/08/21
11:11:12 1619358
+++ webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/util/WSSecurityUtil.java2014/08/21
11:12:58 1619359

@@ -24,6 +24,7 @@
...
+import org.apache.wss4j.dom.WSDocInfo;
...

-    public static void verifySignedElement(Element elem, Document doc, Element securityHeader)
+    public static void verifySignedElement(Element elem, WSDocInfo wsDocInfo)
         throws WSSecurityException {
-        final Element envelope = doc.getDocumentElement();
-        final Set<String> signatureRefIDs = getSignatureReferenceIDs(securityHeader);

           ...
```

old revision
new revision
start line index and number of lines of the old, and new revisions
added line is preceded by a `+`
deleted line is preceded by a `-`

Figure 6.17: Diff output for WSS4J r1619358 and r1619359

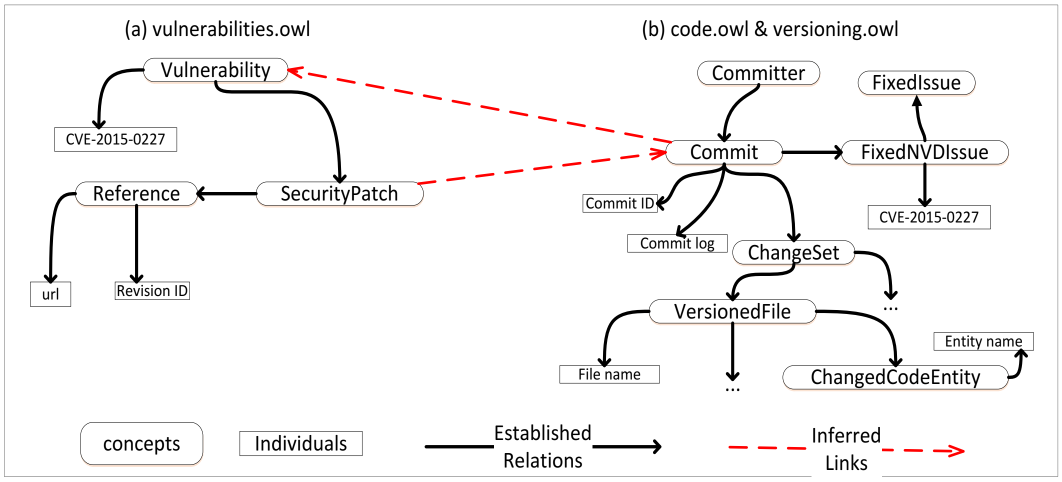


Figure 6.18: Inferred links between vulnerabilites.owl, code.owl, and versioning.owl

```
SELECT ?project ?vulnerablecode ?client ?code
WHERE {
  ?project rdf:type sbson:BuildRelease.
  ?project code:containsCodeEntity ?vulnerableCode.
  ?vulnerableCode  rdf:type sevont:VulnerableCode.
  ?client code:containsCodeEntity ?code.
  ?client sbson:hasDirectDependencyOn ?project.
  ?code main:dependsOn ?vulnerableCode.
}
```

Figure 6.19: Query to retrieve vulnerable code fragments across project boundaries

Given our populated ontologies, we infer a similarity link between instances of a vulnerable product (e.g., Apache WSS4J 1.6.16) in SEVONT and SBSON, as well as links between the vulnerability patch reference (CVE-2015-0227) and the commit containing the patch (modeled in SEON) using the rules in Figures 6.11 and 6.9, respectively. Given these inferred links (Figure 6.18) and using the SPARQL query in Figure 6.19, we can now further restrict our transitive dependency analysis to include only those components that have an actual call dependency to the vulnerable source code.

Findings: Table 6.9 summarizes the results from our third case study, which we performed for CVE-2015-0227. We report on the (manually verified) results obtained from executing our

SPARQL query (Figure 6.19). Table 6.9 shows that 15 of the 242 crawled dependent projects actually use the API from the vulnerable project. This highlights that there are still many systems (6.19%) that rely on libraries with known security vulnerabilities. Moreover, 10 of these 15 dependent projects not only include the API but also actually call the class WSSecurityUtil, which contains the vulnerable code. However, it should be noted that for our specific case study, none of the projects actually called and executed the vulnerable method (verifySignedElement) within the WSSecurityUtil.

Table 6.9: Case Study #3 Results

Project	# Crawled Dependencies	# Actual usage	# Vuln. Classes usage	# Vuln. Methods usage
Apache WSS4J 1.6.16	242	15	10	0

To evaluate if our approach is actually capable of correctly identifying calls to vulnerable methods, we conducted an additional controlled experiment. For this experiment, we manually seeded a method call in Apache CXF-bundle 2.6.15 that invokes the vulnerability in Apache WSS4J API. We downloaded the source code for Apache CXF-bundle 2.6.15 and modified its org.apache.cxf.ws.security.wss4j.policyhandlers package. Figure 6.20 shows the partial class diagram of the modified packages. We modified the includeToken method of the AbstractBindingBuilder class to include a direct call to the vulnerable WSSecurityUtil.verifySignedElement method. We also added the SVAFSymmetricBindingHandler and SVAFAsymmetricBindingHandler to extend SymmetricBindingHandler and AsymmetricBindingHandler to be able to see if our approach also supports the transitive call dependency analysis correctly. We then re-populate the source code ontologies with the new (modified) code facts and invoked the same query we used earlier in the case study.

The results of this query are shown in Table 6.10 and include the classes of our modified project that either directly or indirectly invoke the vulnerable method WSSecurityUtil.verifySignedElement. For sake of simplicity and readability, we only include public and protected methods in the result set. From the reported results, we observed that the vulnerability introduced in AbstractBindingBuilder.includeToken propagates through several methods. For example, the doSVAFAction method is indirectly affected due to its usage of the getSignatureBuilder method. Since SVAFAsymmetricBindingHandler extends

AbstractBindingBuilder and overrides the getSignatureBuilder method, the invocation of doSVAFAction by test2 is correctly identified as a non-vulnerable method call since it does not propagate to the vulnerable WSSecurityUtil.verifySignedElement method.

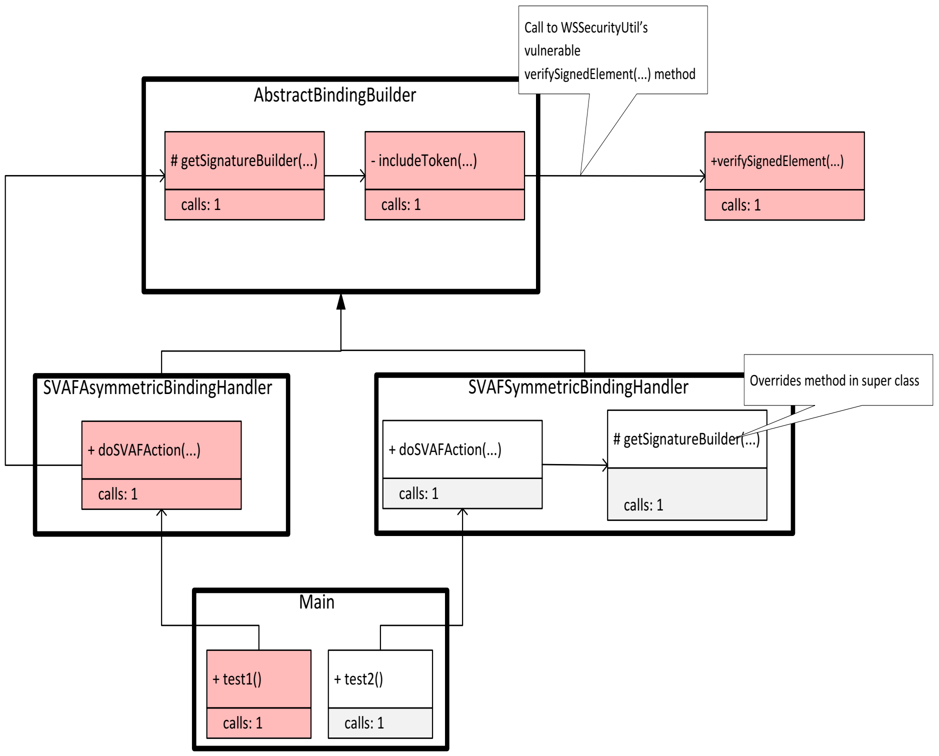

Call to WSSecurityUtil's vulnerable verifySignedElement(...) method
AbstractBindingBuilder
# getSignatureBuilder(...)
calls: 1
- includeToken(...)
calls: 1
+verifySignedElement(...)
calls: 1
SVAFAsymmetricBindingHandler
+ doSVAFAction(...)
calls: 1
SVAFSymmetricBindingHandler
+ doSVAFAction(...)
calls: 1
# getSignatureBuilder(...)
calls: 1
Overrides method in super class
Main
+ test1()
calls: 1
+ test2()
calls: 1

Figure 6.20: Class diagram for our modified package

Table 6.10: Results of Direct and Indirect Usage of the Vulnerable "Wssecurityutil.Verifysignedelement" Method

Class	**# Indirect Vulnerable Methods**	**Indirect Vulnerable Methods**
AbstractBindingBuilder.java	4	handleSupportingTokens(.SupportingToken,boolean,Map, Token, Object)
		getSignatureBuilder(TokenWrapper, Token, boolean, boolean)
		getSignatureBuilder(TokenWrapper, Token, boolean)
		doSVAFAction()
Main.java	1	test1()

6.5 Discussion and Related Work

6.5.1 Comparison Against Existing Tools

As part of our evaluation, we further compared our approach against existing tools that detect known security vulnerabilities in source code across project boundaries. For this comparison, we consider the open source OWASP Dependency-Check (DC) tool and a closed-source tool from SAP labs [132]. OWASP DC performs a static dependency analysis to determine if libraries with known vulnerabilities are included in an application. During the analysis, the tool collects information about the vendor, product, and version. This information is used to identify the Common Platform Enumeration (CPE). If a CPE is identified, a listing of associated Common Vulnerabilities and Exposure (CVE) entries are reported. The SAP tool relies on a dynamic source code level analysis to identify if a vulnerable piece of code is either used directly or indirectly. The tool uses execution traces which are collected after instrumenting the project code including all bundled libraries. Since we did not have direct access to the SAP tool, we replicated their experiments to compare our results with the ones reported in [132].

Given that the OWASP DC tool does distinguish whether a vulnerable library code is used or not, we limit our comparison to: "ide*ntify if a project depends on libraries with disclosed vulnerabilities independent of the use of the vulnerable source code"*. Table 6.11 reports the results from our comparison, which include true positives (TP), false negatives (FN), false positives (FP), and true negatives (TN). The results show that for CVE-2013-2186, both our approach and OWASP DC did not report the vulnerable API. This miss is due to the fact that NVD did not include FileUpload 1.2.2 in the list of affected products. The vulnerability, however, is reported in several JBoss projects which make use of the DiskFileItem class in Apache FileUpload. Our approach currently models only products explicitly mentioned to be affected in NVD. OWASP DC reported CVE-2014-9527 as a vulnerability in POI 3.11 Beta 1. A manual inspection of the patch showed that the class "org.apache.poi.hslf.HSLFSlideShow" contains the patch for the vulnerable code but is not used by "poi-3.11.beta1.jar". Instead, this patch is distributed as part of the POI-HSLF component. For the vulnerability CVE-2013-0248, the patch is located in the default configuration file "using.xml" and the comment of the Java class "DiskFileItemFactory" (but not any executable code). As a result, the SAP tool does not identify the archive as being affected by vulnerable code.

Table 6.11: Comparison of Analysis Results

Vulnerability	Library	SV-AF	SAP tool	OWASP DC
CVE-2014-0050	Apache FileUpload 1.2.2	TP	TP	TP
CVE-2013-2186		**FN**	TP	**FN**
CVE-2013-0248		TP	**FN**	TP
CVE-2012-2098	Apache Compress 1.4	TP	TP	TP
CVE-2014-3577	Apache HttpClient 4.3	TP	TP	TP
CVE-2014-9527	Apache POI 3.11 Beta 1	TN	TN	**FP**
CVE-2014-3574		TP	TP	TP
CVE-2014-3529		TN	TN	TN

As our case studies illustrate, our ontology-based knowledge modeling approach (SV-AF) can integrate information originating from different heterogeneous knowledge resources. In what follows, we discuss how our approach overcomes several challenges identified with both OWASP and SAP tools.

Data integration challenges. Vulnerability and dependency management make use of different naming schemes and nomenclatures: There exist many language-dependent approaches for referencing entities, making the linking of entities across knowledge resources often a difficult task. Consider the following example: Mapping the Spring Core 4.0.3.RELEASE between Maven and NVD. Maven GAV identifier represents this component as groupId=org.springframework; artifactId=spring-core; version=4.0.3.RELEASE. While the CPE for the same component in NVD is: vendor=pivotal; product=spring_framework; version=4.03.

Such identifier naming inconsistency is difficult to resolve during automatic mapping between GAV identifiers in Maven with their corresponding CPE in NVD. For example, the vendor in our example should be Pivotal and not springframework. While a human can easily recognize such correct mapping, this is not the case for an automated solution. Both OWASP DC and the SAP tool compute the SHA-1 of the archives and perform a lookup in Maven central to address this problem. While such a look-up can improve the recall (number of correct mappings found), it also introduces many false positives and false negatives, which affects the accuracy of these tools. Moreover, both tools are limited in their ability to match vulnerabilities and CPEs, making them not only prone to errors but also limit the scope of the analysis to direct dependencies. In contrast, our approach addresses these challenges by taking advantage of the PSL alignment framework. This eliminates the need for one-to-one assignments and establishes weighted links between instances of different modeled ontologies for different data sources.

Moreover, our semantic approach takes advantage of semantic reasoning to infer transitive dependencies.

Flexibility. While dynamic run-time information (traces) can improve precision (SAP tool), the recall will depend on the quality and coverage achieved by these traces. Furthermore, the SAP tool focuses on intra-project analysis, whereas our approach also supports inter-project analysis. As we further show in our case study, by taking advantage of automated reasoning we are able to infer sub-properties (subsumption) and transitive closure dependencies, we can transform often complex and proprietary source code analysis tasks into simpler and more flexible SPARQL queries. For example, the <code:isSubClassOf>, <code:isSubInterfaceOf>, <code:invokesMethod>, and <code:invokesConstructor> are all sub-properties of the transitive <main:dependsOn> property. As such, a simple query (Figure 6.19), can now identify all code entities that transitively depend on a given vulnerable code entity independent of the type, method invocations or inherited classes/interfaces (via subsumption). As we have shown in our controlled study, vulnerable classes can create a backdoor (e.g., through inheritance) to allow for the invocation of vulnerable methods, if these methods are not overridden within the client. With the growing popularity of using 3rd-party APIs [133], the risk of such transitive vulnerable method invocations increases.

Information silos challenges. Although both analysis tools, SAP and OWASP DC are linking different data sources, these resources remain in both approach information silos. They still lack the standardization, knowledge sharing, and analysis result integration required to make them true information hubs. In contrast, our approach introduces a unified standardized representation using ontologies, which supports seamless knowledge integration, interoperability and sharing even on a global scale. The triplestore not only provides persistence of the data but also provides scalability and the use unique resource identifiers (URIs), eases the integration with other knowledge resources, even at a global scale.

6.5.2 Threats to Validity

Internal Validity. An internal threat to the validity is that the experiments rely on our ability to mine facts from the Maven Central and NVD repositories to populate our ontologies. A common problem with mining software repository is that repositories often contain noise in their data due to ambiguity, inconsistency or incompleteness. This threat can be mitigated in our research context, since vulnerabilities published in NVD are manually validated and managed by security experts and therefore making this data less prone to noise. Similarly, the Maven repository captures dependencies related to a particular build file, while ensuring that the dependencies are fully specified and available, eliminating ambiguities and inconsistency at the project build.

Another internal validity threat is that the instance pair matches for our training set were manually created and could potential be prone to human errors. In order to mitigate this threat, we conducted a cross validation of the annotation, where the links were evaluated by another person. Finally, the size of the dataset used to evaluate our approach might not be considered large enough. To mitigate this threat, we evaluated our approach on different dataset sizes to study the effect of the dataset size on our results. As shown in Table 6.12, we observed an average standard deviation between 0.04 and 0.09, for the precision and recall respectively, across all data set sizes. As our results show, the size of the dataset has no adverse effect on the precision and recall of our approach.

Table 6.12: Dataset size evaluation

Data Points	**SV-AF (w=0.1)**			
	Precision	**|Distance from σ|**	**Recall**	**|Distance from σ|**
50	0.76	0.11	0.38	0.26
100	0.87	0.00	0.62	0.02
150	0.88	0.01	0.69	0.05
200	0.9	0.03	0.69	0.05
250	0.89	0.02	0.68	0.04
300	0.86	0.01	0.63	0.01
350	0.87	0.00	0.66	0.02
400	0.87	0.00	0.68	0.04
450	0.88	0.01	0.67	0.03
500	0.88	0.01	0.68	0.04
Avg:	0.87	-	0.64	-
SD (σ)	0.04	-	0.09	-

Other threats to the mining of these repositories are related to the fact that we only extracted vulnerabilities reported from 2010 to 2016 from the NVD database. Given the number of vulnerabilities we extracted for this time period form the NVD database, the dataset is large enough to avoid any bias towards certain vulnerabilities or affected libraries.

External Validity. In terms of external threats to validity, the presented experiments might not be generalizable for non-MAVEN projects. This threat can be partially mitigated through our modeling approach. Given that our modeling approach is based on different levels of abstraction, we also consider and abstract common aspects of the domain of build repositories in our knowledge model. We do model the domain of build repositories as a domain of discourse in the domain-specific layer of our knowledge model. Another external threat to validity for our research is that our evaluation has mainly focused on a quantitative analysis of the results from the case studies, limiting our ability to generalize the applicability and validity of the approach. In order to mitigate this threat, an additional qualitative analysis has to be performed in the form of user studies, which will allow for an evaluation of both, the applicability of the approach and the analysis of the result sets from an expert user perspective.

6.5.3 Related Work in Tracking Known Security Vulnerabilities

Although several approaches for static vulnerability analysis and detection in source code exist (e.g., [132], [134], [135], [136]), there is a lack of tools in identifying and tracking security vulnerabilities on a global scale. Tracking known security vulnerabilities through the Web is different from tracking security defects within the source code of projects. Mitropoulos et al. [137] and Saini et al. [138] use FindBugs one of the known static analysis tool, to find major security defects in Java source codes. The collected information was used in studying the evolution of security-related bugs in a project. In comparison, their approach finds security defects in the source code, while our finds the usage of known security vulnerabilities in software components on a global scale.

Plate et. al [132] proposed a technique that supports the impact analysis of vulnerabilities based on code changes introduced by security fixes. Their approach relies on a dynamic analysis to determine if a vulnerable code was executed within a given project. In contrast, while less precise, we provide a more holistic approach, which not only considers all possible executions but also supports a more general intra and inter-project dependency analysis. We also take advantage of semantic reasoning services to infer implicit facts about the vulnerable code usages within the system, to support bi-directional dependency analysis – including both impacts to external dependencies and vice versa.

Nguyen et. al [131] proposed an automated method to identify vulnerable code based on older releases of a software system. Their approach scans the code base of each prior version for code containing vulnerable code fragments. In contrast, our approach takes advantage of multiple knowledge resources, providing a greater flexibility in the analysis.

Mircea et al. [21] introduce in their Vulnerability Alert Service (VAS) an approach that notifies users if a vulnerability is reported for software systems. VAS depends on the OWASP Dependency-Check tool.

Eventually, analyzing software project artifacts on the security detection level has recently become a very active area of research. Such analysis has valuable results when it comes to finding security-related discussions [139], identifying security/non-security bug reports [140],or predicting vulnerable software components [141]. However, to the best of our knowledge, no research has been conducted on creating an infrastructure of semantic linking between identified security vulnerabilities in traditional software repositories and the security issues listed in software security repositories.

6.6 Chapter Summary

This chapter presented SV-AF, a vulnerability analysis framework based on the integration of our SBSON ontology and a software vulnerability ontology (SEVONT). SV-AF provides developers with an API-level analysis of the impact of vulnerabilities within their projects and global dependencies. Using multi-layers of abstraction, our modeling approach can not only provide a generic analysis approach but also supports the seamless integration of other knowledge resources in the SE domain. This formal knowledge representation allows us to take advantage of inference services provided by the SW, providing additional flexibility compared to traditional proprietary analysis approaches.

In the next chapter, we present another application of SBSON – an API trustworthiness assessment framework based on the SBSON, SEON, and SEVONT ontologies used in this chapter. In addition, the trustworthiness framework in the next chapter also uses an existing software licensing ontology (MARKOS) to provide support for license violation detection.

Chapter 7

7 API Trustworthiness: An Ontological Approach for Software Library Adoption

7.1 Introduction

Most of today's software projects increasingly depend on the usage of external libraries, which allows software developers to take advantage of features provided by Application Programming Interfaces (APIs) without having to reinvent the wheel. Unfortunately, even though third-party libraries are readily available, developers are faced with new challenges with this new form of code reuse, such as being unaware of the existence of libraries, selecting the most relevant library among several possible alternatives, and how to use features provided by these libraries [14], [142].

Several software library recommendation approaches have been proposed to address these challenges. These approaches fall into two main categories: (1) recommendation systems for libraries and APIs based on characteristics such as popularity [133], frequency of migration [16], [143], and stability [117], without considering the context of use of these libraries; and (2) techniques that take a client's context into account when recommending libraries (e.g., using the history of method usages by developers [144]).

However, reused software libraries should not only satisfy a client's functional requirements; they must also satisfy non-functional requirements (NFR) such as security, safety, and dependability [145], which are critical to the success of software systems. NFRs are often referred to as system qualities and can be divided into two main categories: (1) execution qualities- qualities which are observable at runtime (e.g., performance and usability); and (2) evolution qualities, such as testability, trustworthiness, maintainability, extensibility, and scalability, which are embodied in the static structure of a software system. NFRs often play a critical role in the acceptance and trust users will have in a final software product. However, assessing and evaluating the trustworthiness of today's software systems and software ecosystems remains a challenge due to issues ranging from a lack of traceability among software artifacts to limited tool support.

Trustworthiness is also a subjective and ubiquitous term since its interpretation depends on the assessment context of the stakeholder, which might be different among stakeholders and the context of use in which the library is used. Assessment models, therefore, should provide the flexibility and customizability to take into account such specific application contexts and the particular assessment needs of stakeholders [9].

In our prior works, we introduced our Software Build System Ontology (SBSON) and Security Vulnerabilities Analysis Framework (SV-AF) semantic modeling approaches which model the dependencies between OSS libraries and establishes traceability links between security and software databases such as build repositories and version control repositories. The work in this chapter is a continuation of these previous works. In this chapter, we present our Ontology-Based Trustworthiness Assessment Model (OntTAM), an instantiation and extension of the SE-EQUAM assessment model [9], for the domain of software library trustworthiness. SE-EQUAM is a generic quality assessment model which uses ontologies to model and conceptualize quality factors, sub-factors, attributes, measures, weights, and relationships used to assess software quality.

More specifically, we illustrate how OntTAM can be instantiated to take advantage of our existing unified knowledge representation of different Software Engineering related knowledge resources and support an automated analysis and assessment of trustworthiness quality attributes of libraries. We argue that ontologies not only promote and support the conceptual representation of knowledge resources in software ecosystems but also let us take advantage of semantic reasoning during the assessment of trustworthiness quality factors. Furthermore, our modeling approach allows for the customization of the trustworthiness assessment model to reflect specific assessment needs while at the same time facilitates the comparison of trustworthiness across projects, by defining a standard set of measures and sub-factors. In addition to supporting our existing analysis of the impact of API breaking changes and vulnerabilities, OntTAM supports new semantic analysis for software license compatibility.

Our research is significant for several reasons:

- We introduce OntTAM, a novel trustworthiness assessment model that takes advantage of both, our previous generic SE-EQUAM software assessment model [9] and our unified ontological knowledge representation of different SE related knowledge resources [9], [123], [124] while supporting the customization of the model to meet a stakeholder's assessment needs.
- We extend the MARKOS license ontology [10] with semantic rules for three categories of license violations.
- We introduce as part of OntTAM, novel trustworthiness measures, which measure API breaking changes, security vulnerabilities, and license violations. These measures take advantage of our ontologies and semantic reasoning services to allow for a trustworthiness analysis across the boundaries of individual artifacts and projects.
- We report on a case study that illustrating how our approach can be applied to assess the trustworthiness of OSS libraries and discuss the potential impact of these libraries on the trustworthiness of the overall system.

7.1.1 Motivating Example

In what follows, we introduce a motivating example (Figure 7.1) describing how our fictional software developer (Bob), attempts to re-use external libraries while facing several challenges during selecting the best library for his project while reducing their negative effect on the trustworthiness of his own project.

Bob is currently developing an application which requires an embedded database. Bob tries to reduce his development effort, by searching the Internet for possible third-party libraries and components which meet his work context. His search returns Apache Derby[44], an open source embedded DBMS implemented entirely in Java. However, Bob is faced now with the dilemma of deciding upon which version of Derby he should be using – the most recent (Derby version 10.11.1.1) or the most widely used one (Derby version 10.1.1.0). Following the

[44] db.apache.org/derby/

recommendations published in the existing research (e.g., Mileva et al. [2]), Bob decides to use an older version of Apache Derby (version 10.1.1.0) due to its widespread usage/popularity. However, this recommendation results in the reuse of a component, which contains three known security vulnerabilities that are already reported in the National Vulnerability Database (NVD) (see Table 6.1 in Section 6.1.1). In contrast, the newer version of Derby (version 10.11.1.1) does not contain any known vulnerabilities.

However, this is not the only risk Bob is susceptible to when selecting a library. Derby is licensed under the Apache 2 copyright license; for Bob not to introduce any license violation or incompatibility, he must make sure that the selected library is compliant with his project license. For example, one cannot combine code released under the Apache 2 license with code released under the GNU GPL 2 [146].

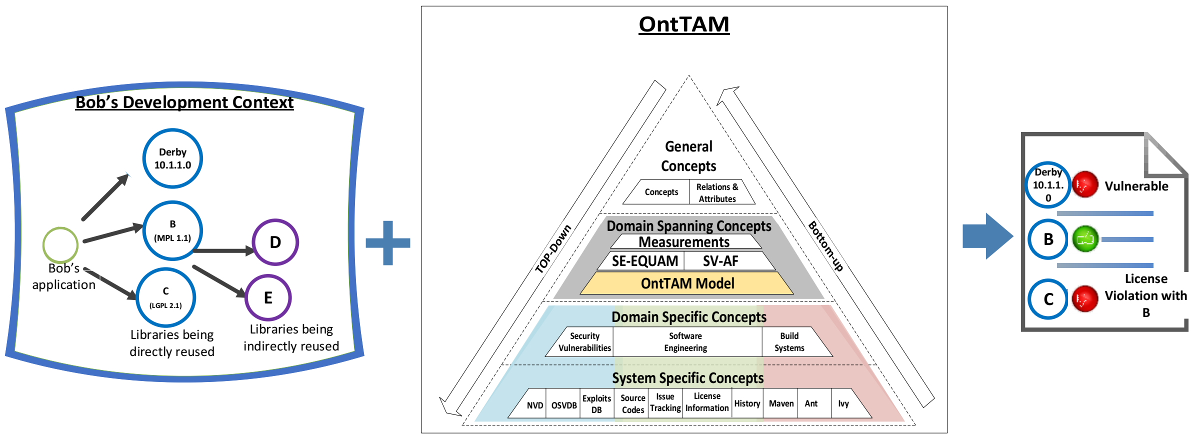


Figure 7.1: Motivating Example – How OntTAM can assist developers in trust assessment

As this example illustrates, several quality-related issues with the reuse of third-party library can arise and they are often difficult to discover by the user, since the relevant information is spread across multiple knowledge resources. The problem is further exacerbated by the large number of additional transitive dependencies which are introduced by these third-party libraries and their dependencies. A vulnerability or license violation might not occur directly between Bob's project and the Derby library, but also between Bob's project and one of the libraries the Derby library depends on.

Note: An earlier version of the work done in this chapter is accepted for publication in the *Software Quality Journal,* 2019 [147].

7.2 Background

The work presented in this chapter combines different knowledge sources (build and dependency repositories, vulnerability databases, source code changes, versioning history, and software licenses) and existing models (SE-EQUAM, SEVONT, and MARKOS). In this chapter, we provide a brief background on Open Source Software (OSS) licenses and the existing MARKOS and SE-EQUAM models we reused. For an overview of the core concepts related to build and dependency management, source code changes, or the SEVONT model, we refer the reader to Sections 3.4, 5.2.1, and 6.2.3 respectively.

Table 7.1: Ten common open source licenses and their traits

License	Requires Attribution[45]	Derivative Work Requirements	
		Public Source Code	Same License
Apache 2	YES	NO	NO
Artistic 2	YES	NO	NO
BSD[46] – Berkely Software Distribution License	YES	NO	NO
EPL 1 – Eclipse Public License	YES	YES	YES
GPL 2 – GNU General Public License	YES	YES	YES
GPL 3	YES	YES	YES
LGPL 2.1 – GNU Lesser General Public License	YES	YES	YES
LGPL 3	YES	YES	YES
MIT – Massachusetts Institute of Technology License	YES	NO	NO
MPL 2 – Mozilla Public License	YES	YES	YES

7.2.1 Open Source Licenses

An OSS license is a legal instrument that allows the creative work (source code) to be used, modified and/or shared under defined terms and conditions [148]. OSS licenses are categorized as either restrictive or permissive. Restrictive licenses (also known as copyleft licenses) require derivative works to be licensed under the same terms. A derivative work is defined as any work

[45] "Requires Attribution" generally means posting in your software's credits, the title of the OSS project, and a copy of its license (with the optional posting of the author, and a link to the project's website).
[46] BSD can refer to a handful of variations on the same license. For the purposes of this work, the common 2-and 3-clause variants are used.

that stems or is adapted from the original work [149]. An example of a restrictive license is the GPL 3. Permissive licenses on the other hand have fewer requirements on derivative works; for example, the MIT License only requires author attribution and reproducing the license with the disturbed software. Table 7.1 lists the ten most frequent licenses with a summary of their pertinent features [150].

7.2.2 License Violations

While dependency management tools have been introduced to automate the downloading and importing of libraries into projects, these libraries still originate from various authors and come with a plethora of OSS licenses (horizontal increase). One library can utilize another library, leading to hierarchies of libraries and license dependencies. All these libraries' licenses must be compatible and compliant with each other. License violations and incompatibilities are an often-overlooked factor when recommending licenses and therefore can significantly impact the trustworthiness of software systems. When incompatible licenses are used together, a license violation occurs. A license violation is defined as "*the act of making use of a (licensed) work in a way that violates the rights expressed by the original creator*" [151]. That is, not following the legal terms and conditions set out in the source license. Software authors who commit a license violation open themselves to the possibility of being sued; sometimes this risk can amount to millions of dollars as in the recent case of Oracle v. Google [152].

It should be noted that even though the term license violation is used throughout this thesis, a definitive violation is only determined as such by judge or jury. Consequently, "*potential*" is the operative word when discussing license violations.

7.2.3 The MARKOS License Ontology

In order to find possible license violations, a definition of the rules and permissions associated with a license is needed. In addition, one must outline the allowed and disallowed interactions of any two OSS licenses. Fortunately, the MARKOS ontology [10] provides a formal vocabulary and a set of rules for helping software developers to analyze open source license compatibility issues. Table 7.2 lists the license permissions defined by the MARKOS ontology.

The scope of this research is limited only to the Reciprocity permission type. Reciprocity is important to this research because its context is straightforwardly captured by an ontology and easily relatable to the Maven repository of projects. The reciprocity requirement mainly influences (but is not the sole requirement for) the definition of license compatibility demarcated later in this chapter.

Table 7.2: Permissions defined in the MARKOS ontology.

Permission	Description Attribution
Adaptation	An OSS license allows the original creative work to be adapted and modified.
Distribution	One can publicly distribute the source code.
LibraryUsageWithout Reciprocity	Reciprocity means the source code from a derivative project must be released under the same license as the derived project for both libraries to be used together.
PatentGrant	Some source code algorithms or processes are patented, and the author agrees to grant permission to any downstream user of the source code
Reproduction	One can reproduce or make copies of the source code.
Sublicensing	A user of this source code is permitted (or not) to sublicense the code to anther license

Beyond reciprocity, violations of some of the other permissions are harder to detect because they are violated outside of the realm of a Software Engineering context. For example, the authors of BusyBox sued Samsung in 2009 [153] and settled in 2010 [154] because Samsung was using BusyBox's FLOSS project without publicly publishing the source code (when distributing the software with their hardware). This is a violation of the distribution term of the GPL2 (which would equate to the Distribution permission in the MARKOS ontology). This was only found by manually checking the physical product (in this case a Samsung television) and verifying that the FLOSS was indeed running on the TV hardware. We do not (yet!) have an automated way of testing all the physical products in the world. Therefore, in creating a definition of license violation, we must combine multiple permissions that are feasible to determine. These permissions provide a basis to construct definitions of compatibility, incompatibility, and license violations, which will be further described in the next section.

7.2.4 Evolvable Quality Assessment Metamodel (SE-EQUAM)

Quality is a widely used term to evaluate the maturity of development processes within an organization. Defining quality allows organizations to specify and determine if a product has met certain non-functional and functional requirements. However, as Kitchenham [155], [156] states: "quality is hard to define, impossible to measure, easy to recognize." Unlike functional requirements, where a single analysis technique (e.g., use case modeling) is sufficient to identify essentially all requirements, the same analysis is not appropriate for all quality requirements. Quality, as defined by ISO 9000, is the "degree to which a set of inherent characteristics fulfills requirements", where a requirement is a "need or expectation that is stated, generally implied or obligatory" [157].

Assessing the evolvability of software systems has been addressed in existing research through the introduction of software quality models e.g., McCall [158], ISO/IEC 9126[47], and QUALOSS [159]. These models share a common, while informal (not machine-readable), structural representation of software qualities (Figure 7.2).

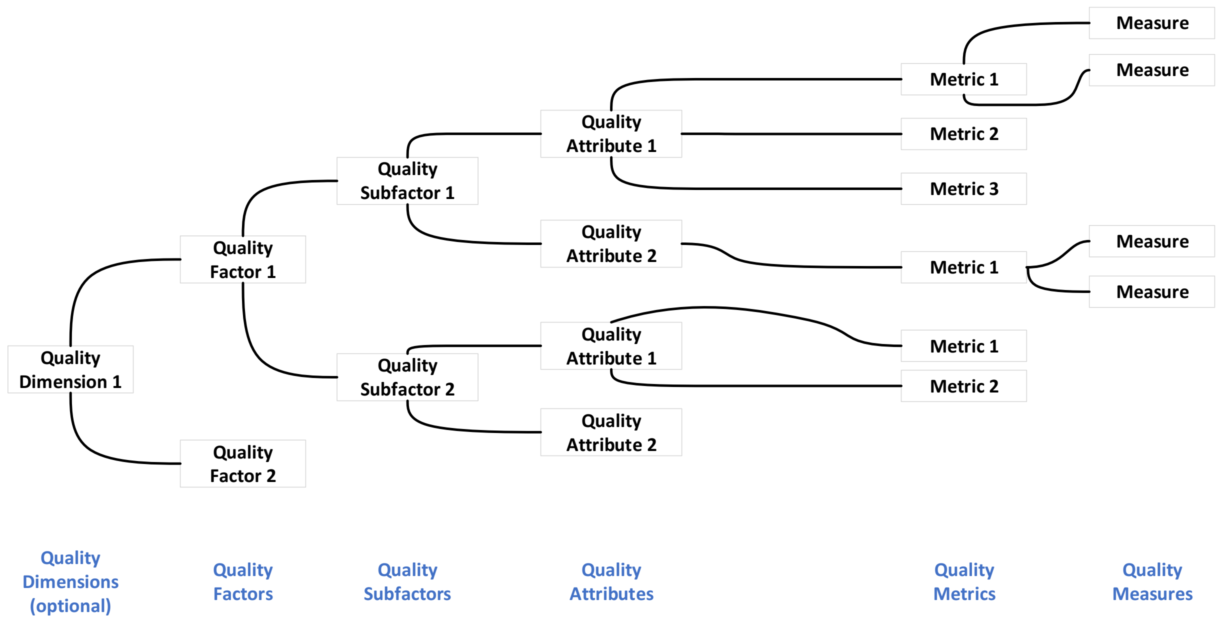


Figure 7.2: Generic structure of quality assessment models [160]

While these models can assess qualities in a given context, they lack the required formalism and semantics to allow them to evolve to meet the modeling requirements of different assessment contexts. The ability to adjust to change assessment needs was the main motivation for SE-EQUAM, an Evolvable QUAlity Meta-model that derives a formal (machine-readable) domain model that can adapt to changes in the assessment needs in terms of both: artifacts being assessed and their assessment criteria [9]. SE-EQUAM addresses these challenges by taking advantage of the Semantic Web and its supporting technologies. SE-EQAM uses ontologies to model and

[47] https://www.iso.org/obp/ui/#iso:std:39752:en

conceptualize quality factors, sub-factors, attributes, measures, weights, and relationships used to assess software quality. Input artifacts for the assessment model are various software artifacts such as version control systems and issue trackers; and its outputs, are quality assessment scores based on the different assessment criteria. Ontologies not only provide a formal way to represent knowledge but also can eliminate ambiguity, enable validation, and provide a consistency-checking approach [161]. SE-EQUAM uses semantic reasoners to infer hidden relationships between domain model attributes. Given its formal representation, SE-EQUAM allows for its reuse by simplifying the instantiation of new domain-specific instances of the model. More details about the semantic reasoning are provided in [9].

Figure 7.3 illustrates the reuse and instantiation of our SE-EQAM model. The generic syntactic meta-model, which is a generic model that forms the basis for all quality models can be instantiated by a domain model (e.g., ISO/IEC 9126). Furthermore, SE-EQUAM allows for a semantic mapping between the syntactical meta-model and a semantic ontology meta-model, which can then be instantiated as domain model ontology based on user-defined assessment criteria.

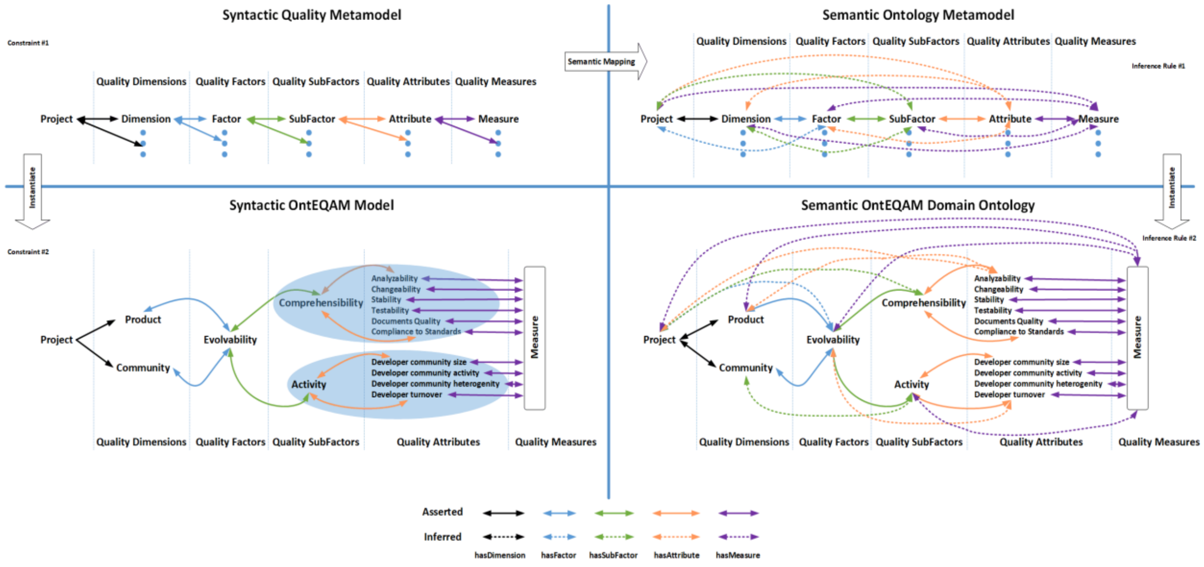


Figure 7.3: SE-EQUAM ontology meta-model reuse to instantiate a domain model ontology (OntEQAM) [9]

The SE-EQUAM Process. The general SE-EQUAM process (Figure 7.4) represents a set of tasks and activities which we followed to allow for deriving a generic quality assessment method that can be used to customize and instantiate the generic model to meet a stakeholder's specific quality assessment context.

The input to the SE-EQUAM process is software artifacts and a set of core quality measurements applicable to these artifacts. In the next step, a common ontological representation for these artifacts has been established by re-using existing models or customizing existing models to meet the requirements of these artifacts. As part of the model adjustment activity, quality metrics and measurements included in the core model can be customized and extended to reflect a specific model context. The output of this process is an instantiated assessment model, which meets specific user and project assessment requirements, by providing a quality assessment at both individual artifact and overall product level. Figure 7.4 illustrates the high-level activities and major tasks involved in the SE-EQUAM instantiation method.

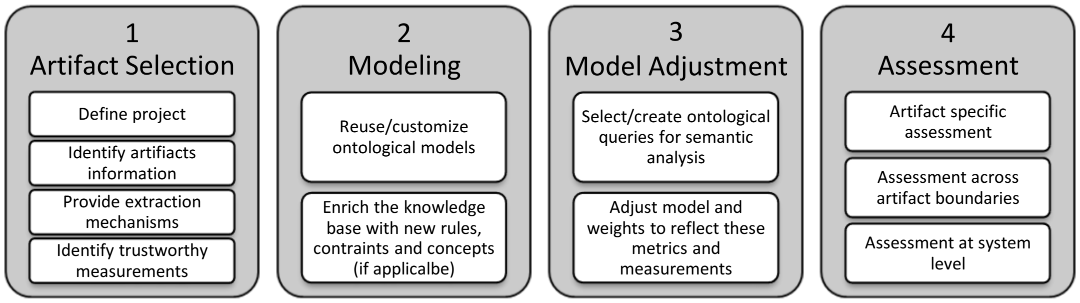

1
Artifact Selection
Define project
Identify artifiacts information
Provide extraction mechanisms
Identify trustworthy measurements
2
Modeling
Reuse/customize ontological models
Enrich the knowledge base with new rules, contraints and concepts (if applicalbe)
3
Model Adjustment
Select/create ontological queries for semantic analysis
Adjust model and weights to reflect these metrics and measurements
4
Assessment
Artifact specific assessment
Assessment across artifact boundaries
Assessment at system level

Figure 7.4: SE-EQUAM Process to instantiate evolvability model

In the next section, we introduce OntTAM, which illustrates a concrete instantiation of the SE-EQUAM process to create a semantically enriched trustworthiness quality assessment model for software libraries.

7.3 Ontology-based Trustworthiness Assessment Model (OntTAM)

OntTAM, an instantiation of the SE-EQUAM [9] ontology meta-model, illustrates how our modeling approach can take advantage of the unified ontological representation of both software artifacts and the generic SE-EQUAM quality assessment model. OntTAM instantiates a domain-specific quality model to assess the trustworthiness of software projects and, more specifically, the trustworthiness of external libraries. OntTAM reuses SE-EQUAM's core quality model structure, which is based on quality factors, sub-factors, attributes, measures, weights, and relationships, and extends them with trustworthiness specific aspects. Inputs to OntTAM are knowledge resources such as version control systems, build systems, project license information, and security vulnerability information. The output of OntTAM is a trustworthiness assessment score for either an individual metric or an aggregation of sub-factors and factors for the overall product/library quality. The model thereby takes advantage of the OWL and RDF/RDFS semantic reasoning capabilities to infer hidden relationships between domain model attributes and to ensure the consistency among these attributes.

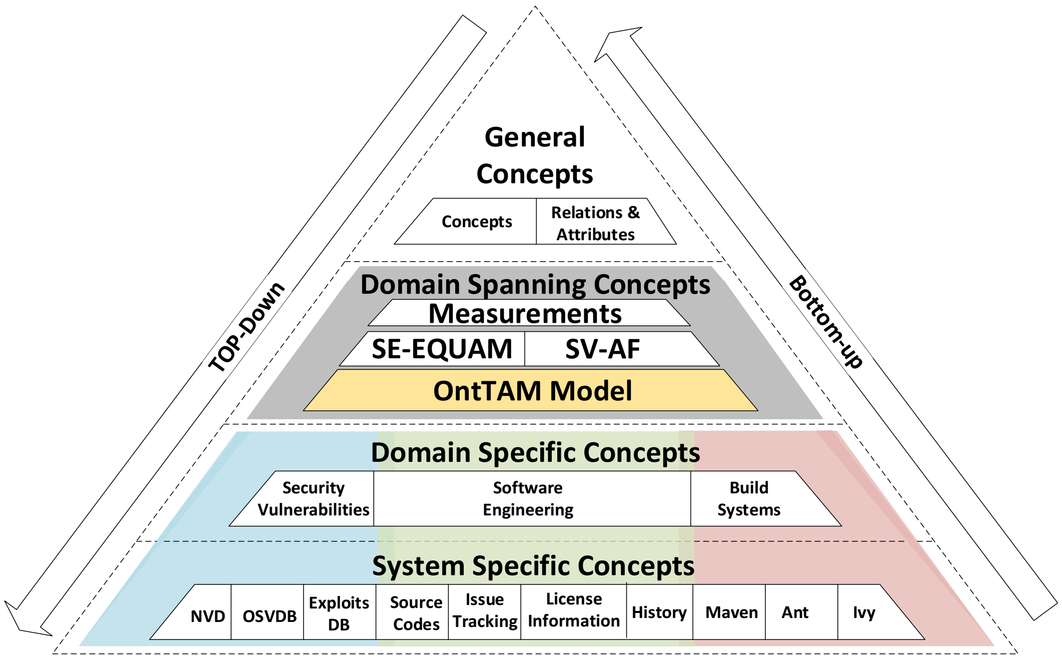


Figure 7.5: The Software Trustworthiness Ontology Hierarchy

Figure 7.5 provides an overview of the knowledge model framework and its organization in terms of ontologies and their abstraction levels. While these ontologies may be derived modeled and used independently, a key objective of our approach is the knowledge integration across ontology boundaries, using both ontology alignments and semantic linking to create a unified ontological knowledge representation.

In what follows, we present our OntTAM methodology to further demonstrate how we instantiate different trustworthiness sub-factors (i.e., security, reliability, and legality), to establish a trustworthiness assessment for OSS products (e.g. external libraries). More specifically, we discuss in detail the four major steps involved in instantiating our customized OntTAM trustworthiness assessment model (see Figure 7.4): artifact selection, modeling, model adjustment, and the assessment process.

7.3.1 Artifact Selection

The input to OntTAM are artifacts relevant to the reuse of software libraries within projects. These software artifacts can be categorized into endogenous and exogenous data. Endogenous data represents data available internally to a software development environment (e.g., software artifacts related to versioning systems, issue trackers, software licenses, and build systems). Exogenous data refers in our context to data available externally to the software development environment (e.g., external vulnerabilities databases). Extracting and populating facts from these artifacts are often based on techniques commonly used by the MSR community [64], [162], [163]. It should be noted that unstructured or semi-structured information (e.g., vulnerability descriptions and license information) often requires several preprocessing steps such as natural language analysis (NLP), as well as data cleansing to improve the quality of the data prior to the ontology population.

7.3.2 Model and Model Adjustment

In this section, we discuss our knowledge modeling process in detail. It should be noted that in order to improve readability, we use prefixes as substitutes to the fully qualified names of our ontologies. The ontology prefixes used in this chapter can be dereferenced using the URIs shown in Appendix A.

7.3.2.1 Modeling Project Trustworthiness

Since OntTAM is based on the generic SE-EQUAM model, OntTAM is an extension and specialization of our core SE-EQUAM software quality assessment model. OntTAM is extended to provide a syntactical trustworthiness quality model that includes and defines a set of sub-factors, attributes, and metrics required for the assessment of trustworthiness. Many of these trustworthiness factors, attributes, and metrics are derived from existing work on trustworthiness assessment of open and closed source projects [9], [160]. The OntTAM specific trustworthiness assessment is based on the two general quality dimensions, the community, and product dimension. The community dimension assesses the adoption of a software product by the community over an extended period, by considering the popularity in terms of downloads, rankings, and activity of the development community. The product dimension assesses the internal structure of the product and the development processes that impact its reusability which is the focus of this paper.

Figure 7.6 provides an overview of the complete model instantiation process which creates as its output a formal (machine-readable) and semantic enriched trustworthiness assessment model. The process involves applying both a syntactic and semantic mapping from SE-EQUAM to OntTAM. While the syntactical model allows us to answer basic queries such as: *What are the sub-factors associated with product trustworthiness*? The semantic mapping enables the use of DL axioms (such as the property chain axiom) to infer new implicit relationships (dashed lines in Figure 7.6– semantic OntTAM ontology) from explicitly modeled relationships in OntTAM (solid lines in Figure 7.6).

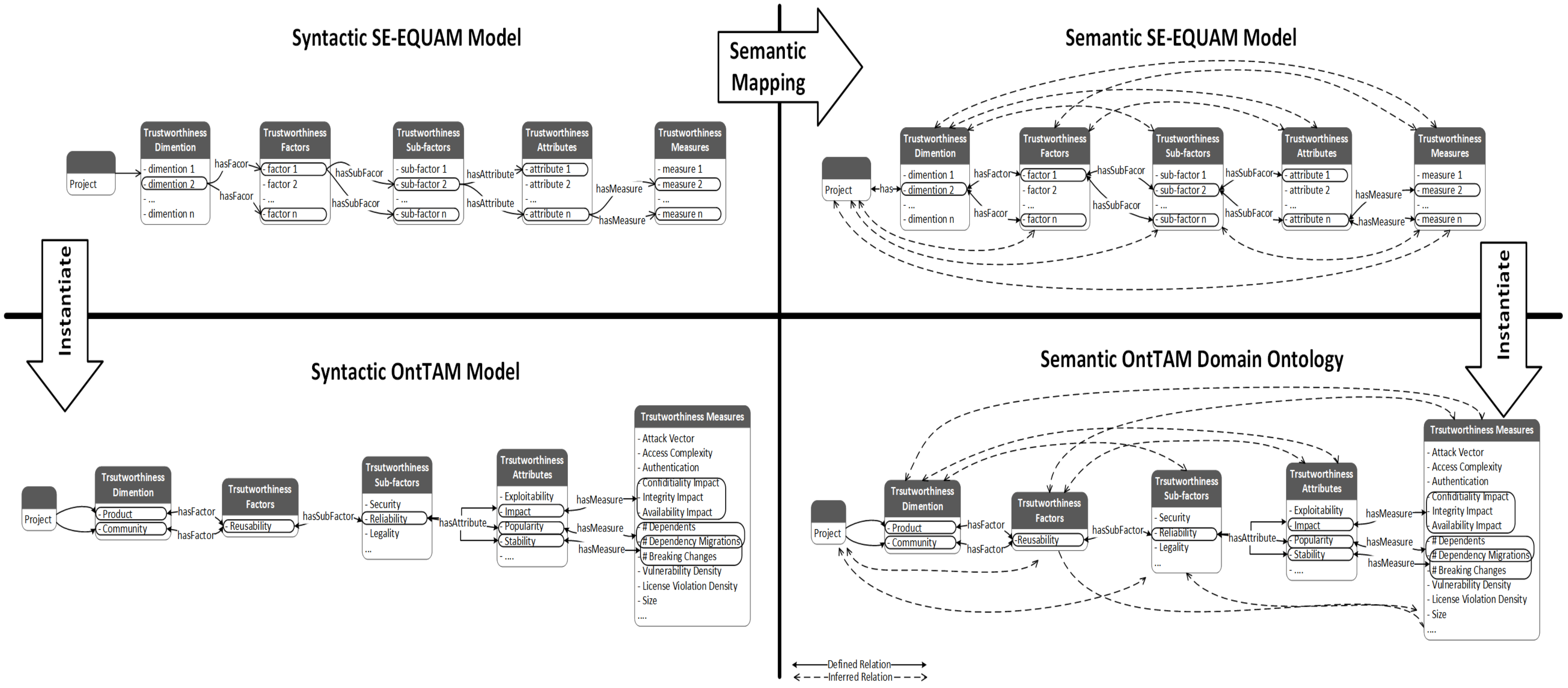


Figure 7.6: Reuse of the SE-QUAM meta-model to instantiate the OntTAM domain model ontology

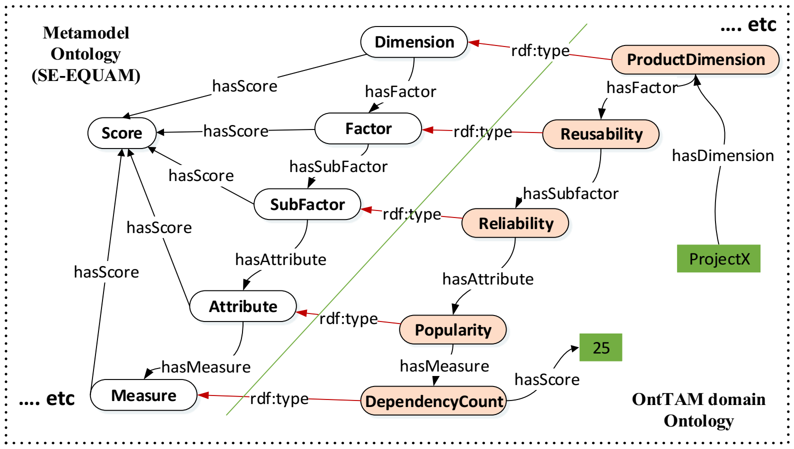


Figure 7.7: An example defining the associated trustworthiness concepts and measures for a sample project

Figure 7.7 illustrates the main steps which are applied to associated trustworthiness concepts and measures for a sample project (ProjectX):

1. Define the product and community dimensions.
   <onttam:ProductDimension><rdfs:type><sequam:Dimension> and
   <onttam:CommunityDimension><rdfs:type><sequam:Dimension>.
2. Define reusability as a factor that is associated with the product dimension.
   <onttam:ProductDimension><sequam:hasFactor><onttam:Reusability> and
   <onttam:Reusability><rdfs:type><sequam:Factor>.
3. Following the same approach, OntTAM defines reliability as a sub-factor of reusability which is associated with the popularity attribute.
   <onttam:Reusability><sequam:hasSubfactor><onttam:Reliability>,
   <onttam:Reliability><rdfs:type><sequam:Subfacor>,
   <onttam:Reliability><sequam:hasAttribute><onttam:Popularity> and
   <onttam:Popularity><rdfs:type><sequam:Attribute>.
4. Assuming that OntTAM assesses a product's reusability through the popularity trustworthy attribute using the DependencyCount measure, we can now define this as:
   <onttam:Popularity><seon:hasMeasure><sbson:DependencyCount> and
   <sbson:DependencyCount><rdfs:type><seon:Measure>.

Finally, we enrich OntTAM's syntactical model to become a semantic model, by establishing additional semantic relationships by adding property chain axioms (e.g., <sequam:hasDimension> relationship with <sequam:hasSubfactor> and <sequam:hasMeasure>). The following are examples of OWL 2 property chain axioms which we added to be able to take advantage of RDFS reasoning during the assessment process.

- Project-related OWL 2 property chain constructs:
    - SubPropertyOf(ObjectPropertyChain(:Project :hasFactor) :Factor)
    - SubPropertyOf(ObjectPropertyChain(:Project :hasSubfactor) :Subfactor)
    - SubPropertyOf(ObjectPropertyChain(:Project :hasAttribute) :Attribute)
    - SubPropertyOf(ObjectPropertyChain(:Project :hasMeasure) :Measure)
- Dimension-related OWL 2 property chain constructs:
    - SubPropertyOf(ObjectPropertyChain(:Dimension :hasSubfactor) :Subfactor)
    - SubPropertyOf(ObjectPropertyChain(:Dimension :hasAttribute) :Attribute)
    - SubPropertyOf(ObjectPropertyChain(:Dimension :hasMeasure) :Measure)
- Factor-related OWL 2 property chain constructs:
    - SubPropertyOf(ObjectPropertyChain(:Factor :hasAttribute) :Attribute)
    - SubPropertyOf(ObjectPropertyChain(:Factor :hasMeasure) :Measure)
- Subfactor-related OWL 2 property chain constructs:
    - SubPropertyOf(ObjectPropertyChain(:Subfactor :hasMeasure) :Measure)

7.3.2.2 Integration with Other Knowledge Artifacts

Assessing the overall trustworthiness of a software library requires us not only to instantiate OntTAM but also to integrate it with other ontological software knowledge artifacts to be able to derive and integrate novel trustworthiness measures. For the integration, we take advantage of software artifact ontologies we have created and refined over the years [123], [164], [165] and by reusing existing ontologies [34] that model different software artifacts. Figure 7.8 provides an overview of the software artifacts and their ontologies which we integrate with OntTAM. These artifacts include, but are not limited to, (a) Software Evolution Ontologies (SEON) which model software engineering repositories such as source code, version control systems, and issue tracker systems, (b) the Build Systems ONtology (SBSON) which captures knowledge about build management systems (e.g., Maven), (c) the Software sEcurity Vulnerability Ontologies (SEVONT) for modeling software security vulnerability information such as severities, impacts, vulnerabilities types, and patch information found in different vulnerability databases, and (d) MARKOS which models software license compatibilities.

The integration of these heterogeneous knowledge resources allows us to introduce different trustworthiness measures related to the reuse of software libraries. More specifically, in this research, we introduce the following three trust criteria: API breaking changes, security vulnerabilities, and license violations. Figure 7.8 shows the core concepts and object properties, distributed across the different abstraction layers of our knowledge modeling framework (Figure 7.5). It should be noted that the omitted data properties to improve the readability of the figure.

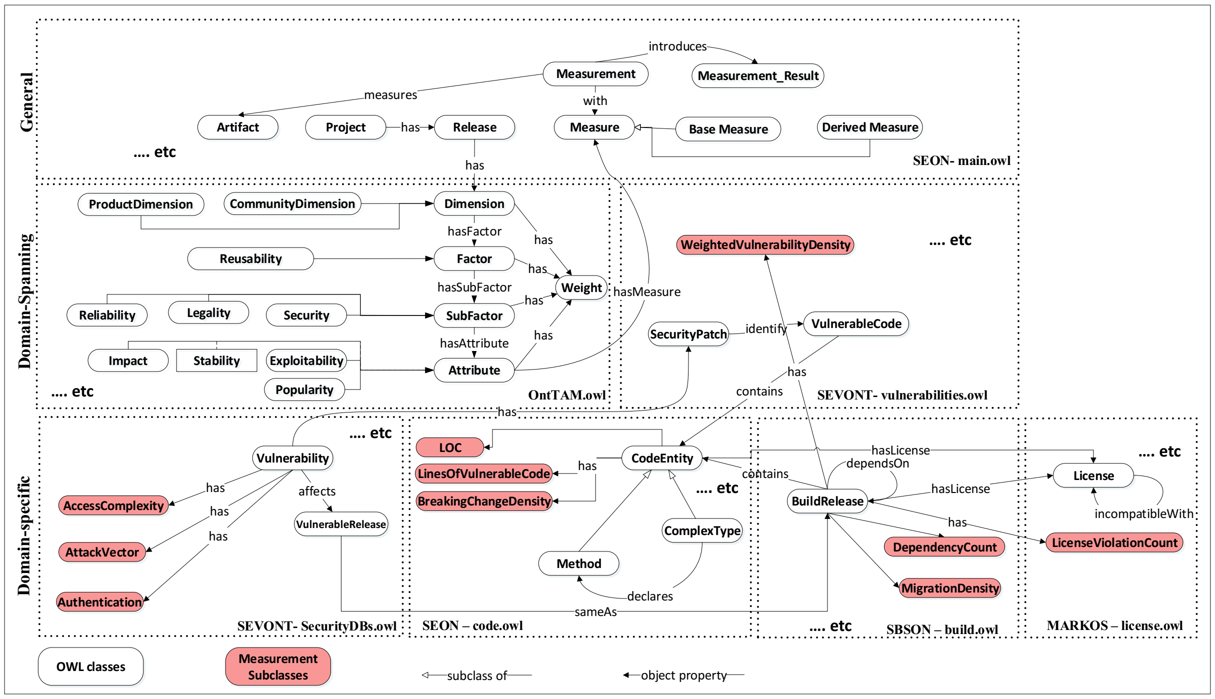


Figure 7.8: Integrating OntTAM ontology into SV-AF model and reusing SE-QUAM concepts

Among the core concepts used from these ontologies is the <sbson:BuildRelease> from the SBSON build ontology, a subclass of the <main:Release> concept, which allows captures the fact that a project can have several releases (including library releases). A release has a <markos:License> and defines its dependencies on other releases. Each release contains a set of <code:CodeEntity> elements such as <code:Field>, <code:Method>, and <code:Class>. A release can be affected by a <sevont:Vulnerability>, leading to the release of a new version containing a <sevont:SecurityPatch>. A security patch corresponds to code changes introduced to fix some existing <sevont:VulnerableCode>, which is part of a <code:CodeEntity>. For example, if a class or method is modified during a security patch, then this code change can be used to locate the original <sevont:VulnerableCode>. The OWL classes, <sevont:SecurityPatch> and <sevont:VulnerableCode>, are linked in our model through an object property. For a complete description of the ontologies, how they are built, the alignment processes, and reasoning, we refer the reader to Chapters 4, 5, and 6.

All these core concepts have metrics used by the OntTAM assessment process. Measures have a unit and are expressed on a scale, e.g. an ordinal or nominal scale. Information about units and scales can be used to perform conversions [34]. Many base measures, such as the number of lines of vulnerable code (LOVC), number of known vulnerabilities, vulnerabilities severities (scores), and number of license violations provide, when viewed in isolation, only limited insights. Additional derived measures are needed to support further analysis and assessment of software artifacts. These derived measures represent an aggregation of values from different subdomains, for example, the number of vulnerabilities per class is an aggregation of measures derived from source code and the vulnerability repositories. While the abstract measurement concepts are defined in the general upper layer of our integrated model (Figure 7.8), many Base Measures (e.g., Size) and Derived Measures (e.g., Weighted Vulnerability Density) are modeled in the domain-specific layer.

7.3.3 Measures and Metrics

An essential feature of our modeling approach is to allow users to customize the OntTAM model through user-defined queries, which might introduce different metrics, ranging from simple metrics to semantic rich metrics queries that take advantage of implicit knowledge inferred by ontological reasoners. Given our ontology-based modeling approach, these analysis results can also be materialized to enrich our knowledge base and to promote reuse of existing analysis results. In what follows, we introduce some metrics to be later used for the assessment of the trustworthiness of systems. These metrics take not only advantage of our unified representation, but also inference services provided by the Semantic Web.

The **Weighted Vulnerability Density (WVD) Metric** compares software systems (or their components) based on severity scores of known vulnerabilities. The objective of WVD is to measure the impact of known vulnerabilities on a product's quality, with the most severe vulnerabilities having the greatest impact. The metric can be applied, for example, to prioritize the patching of vulnerabilities based on their severity. To account for both direct and indirect impacts of vulnerabilities, we introduce the WVDdirect and WVDinherit measures. Although a project can have a WVDdirect score of 0 since no known security vulnerability has been reported for the core project, it is still possible that the project is exposed to indirect vulnerability found in external (third party) dependencies (components) that are included in the parent project. Such a potential security risk will be assessed by the WVDinherit measure.

$$WVD_{direct}(release) = \frac{\sum_{i=1}^{V} w_i}{S} \qquad \text{(Equation 1)}$$

where S is the size of the software (in KLOC), w_i is the weight (severity score) of a known vulnerability affecting the system, and V is the number of known vulnerabilities in the system.

$$WVD_{inherit}(release) = \sum_{i=1}^{n} \left\{ \left(\frac{vulnerable\ APIs\ in\ d_i\ used\ by\ release}{total\ vulnerable\ APIs\ in\ d_i} \right) * WVD_{direct}(d_i) \right\} \qquad \text{(Equation 2)}$$

where n is the number of dependencies used by release, and d_i is the ith dependency.

$$WVD_{overall}(release) = WVD_{direct}(release) + WVD_{inherit}(release) \qquad \text{(Equation 3)}$$

License Violation Count (LVC) is a measure to assess the number of license violations that exist within a given project. This measure can indicate potential long-term risks associated with intellectual rights violations that exist within a project. A license violation occurs if any of the dependent components of a parent project includes components with non-compatible licenses. Open source code license violations are often due to the fact that many software developers are simply neither aware nor well-versed in open source license compliance. For example, in 2008 the Free Software Foundation (FSF) claimed that various products sold by Cisco under the Linksys brand had violated the licensing terms of many programs on which FSF held the copyright[48]. These FSF programs were under the GNU General Public License, a copyleft license which allows users to modify a piece of software as long as the derivative work is under the same license.

[48] https://en.wikipedia.org/wiki/Free_Software_Foundation,_Inc._v._Cisco_Systems,_Inc.

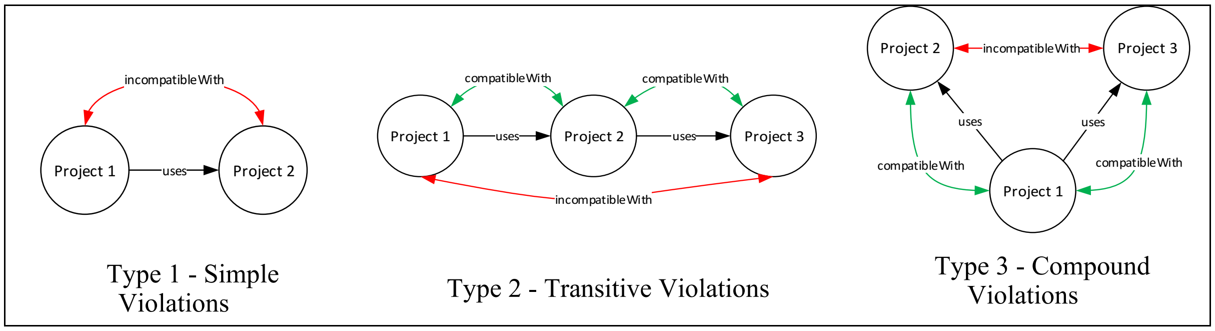


Figure 7.9: Categories of license violations

In this work, we identify three main categories of license violations: simple violations, transitive violations, and compound violations (see Figure 7.9). LVCsimple, LVCtransitive, and LVCcompound are base measures associated with each category. Details on how license violations are identified are presented in Section 7.4.3.

$$LVC_{overall}(release) = LVC_{simple}(release) + LVC_{transitive}(release) + LVC_{compound} \qquad \text{(Equation 4)}$$

Breaking Change Density (BCD) Metric is a normalized measure which represents the ratio between breaking and non-breaking API changes that are introduced in a project. API changes often occur as a project and its components evolve inconsistently, resulting in incompatibilities of APIs and API calls. This measure can be used to determine the stability of an API over time – how often do breaking changes occur. The BCD metric can be represented formally as follows:

$$BCD = \frac{\#\ breaking\ API\ changes}{\#\ nonbreaking\ API\ Changes} \qquad \text{(Equation 5)}$$

Breaking Change Impact (BCI) measures the impact of breaking changes on client applications, by assessing a client application and its use of APIs with a changed contract. The impact of breaking changes on clients can be both direct and indirect. Details on how we identify the impact of breaking changes are presented in Sections 5.4 and 5.5. We introduce two measures that capture both direct and indirect breaking changes

We represent the BCI metrics formally as follows:

$$BCI_{direct}(C, D) = \frac{\#\ breaking\ API\ changes\ in\ D\ used\ by\ C}{\#\ breaking\ API\ Changes\ in\ D} \qquad \text{(Equation 6)}$$

$$BCI_{indirect}(C, < D_1, \dots, \quad D_n >) = \frac{\#\ breaking\ API\ changes\ in\ < D_1, \dots,\ D_n >\ used\ by\ C}{\#\ breaking\ API\ Changes\ across\ < D_1, \dots,\ D_n >} \qquad \text{(Equation 7)}$$

where C is the client project, D is the reused library, and <D1, …, Dn> is the set of (direct and transitive) different library releases being reused by the client.

7.3.4 Assessment Process

Given that stakeholders, with varying contexts, have different assessment needs, our OntTAM assessment process allows for the customization of trustworthiness assessment model in terms of sub-factors and attributes being assessed as well as the individual weights assigned to them. While the default weight for all sub-factors and attributes are equal, users can customize these weights to closely match their assessment objective and context. Furthermore, most existing assessment approaches rely on crisp boundaries (e.g., based on thresholds) which can lead to inaccuracies in the assessment process. It is not always feasible or desirable to use crisp values especially when one deals with values which are close to the boundaries. For example, let us consider an assessment approach with a vulnerability count threshold of 4. Based on this crisp boundary, a project with a reported number of 5 known vulnerabilities will be assessed as being non-trustworthy, even if it can be considered almost borderline to being considered trustworthy. To further exemplify the problem, using the crisp boundary values, there would not be any difference between a project with 5 known vulnerabilities and another project with 100 vulnerabilities, both projects would be considered equally non-trustworthy. This problem does not only occur at the individual measurement level but also at other assessment levels (e.g., sub-factor, factor). To address this challenge, we apply a fuzzy logic assessment and inference approach to eliminate the need for crisp value boundaries.

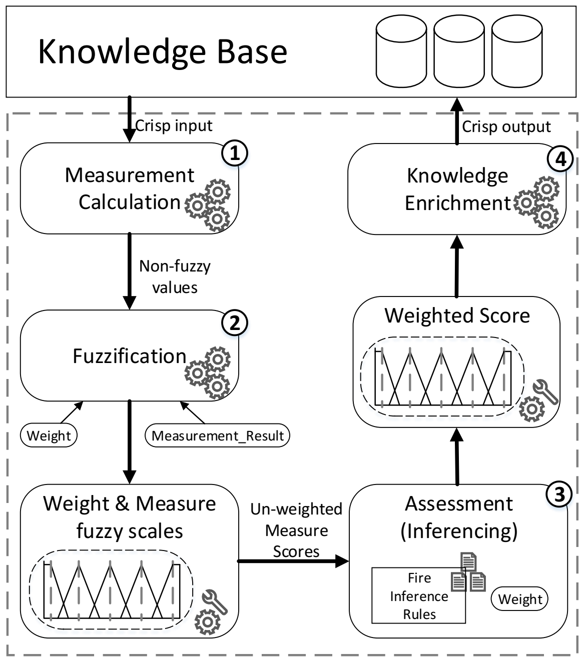


Figure 7.10: Fuzzy Assessment Process Steps

Figure 7.10 shows the set of transformation steps, which are performed during the fuzzification of the assessment process, with details of each step discussed in more details throughout the section.

(1) ***Measure Calculation***: Input to this step are raw values from the populated ontologies. Measures are calculated by querying our populated knowledge base for the base and derived measures introduced in the previous section (e.g. WVD).

(2) ***Fuzzification***: The extracted quality measures and weight values are used to create fuzzy scales in the fuzzification step. As part of the fuzzification step, fuzzy scales are created for the different measures, the assessment weights (provided by stakeholders of the assessment model to assign a level of importance to different measures), and the overall assessment result. These results are converted to linguistic variables, which are variables whose values are expressed as words or sentences (values like e.g., high, not very high, low) [166]. These linguistic variables are the building blocks of Fuzzy Logic and become the input for the fuzzification inference engine.

Figure 7.11 shows an example of a fuzzy scale created for the WVD measure and its assessment weights. The x-axis represents the measurement results range and the y-axis the membership degree (range is 0-1). The higher the membership value, the stronger the measurement’s relation to its fuzzy result scales. The overlap between boundaries of categories in the fuzzy scale demonstrates the uncertainty in interpreting boundary measurement results.

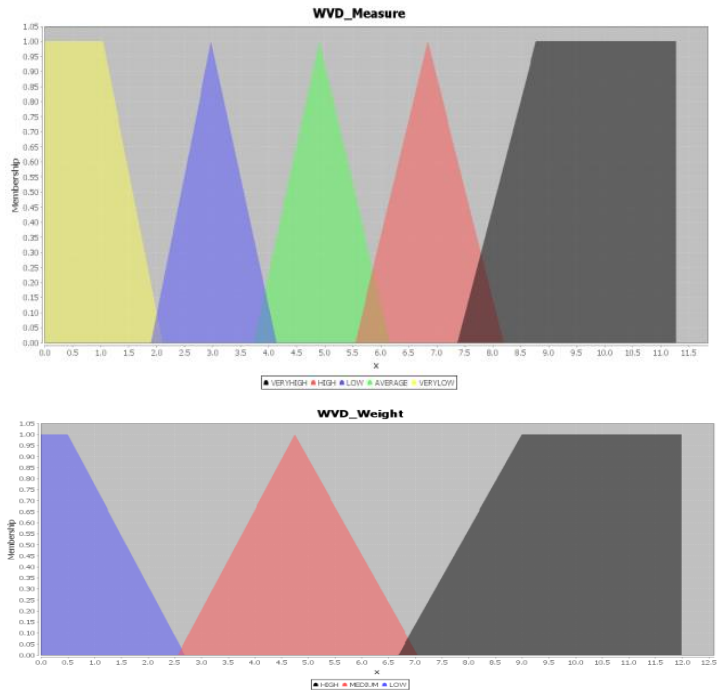


Figure 7.11: WVD measure fuzzy scale and Weight Fuzzy Scale for WVD measure

Since high WVD, LVC, and BCD measures lower the overall quality and trustworthiness score of a project, we made the following three assumptions to automate the fuzzy inference rules for these measures: (a) in cases when the user-specified weight is high then the individual measure score is one level lower, VeryPoor scores will keep their values (e.g., a high weight will change an Excellent score to VeryGood); (b) the opposite holds for low weights, which reflects that their scores are less relevant to the overall assessment their scores are adjusted by one level higher. Excellent scores keep their values; (c) with medium weight, scores keep their values. These assumptions reflect the fact that when a measure is of high importance to the assessment (high weight), its score should be more sensitive to a low measure value.

(3) ***Inference and Assessment***: Input for this step is the fuzzified measure and weight values in the form of linguistic variables. These linguistic results are now transformed into the final assessment score by executing a set of fuzzy inference rules. The de-fuzzification is based on a set of fuzzy inference rules, which are expressed in the Fuzzy Control Language (FCL)[167]

using the JFuzzyLogic inferencing engine [168]. The inference engine fires the relevant fuzzy rules based on the provided input. Firing rules will calculate the final weighted overall measurement result which is a combination of all the different measures. Using the Center of Gravity (COG) method, considered as one of the most popular de-fuzzification methods [169], the overall fuzzy measurement result is de-fuzzified back into a numerical assessment measurement results in order to be populated back to the knowledge base. As part of our assessment, we create a Fuzzy Control Language (FCL) file for each measure. The complete set of FCL files for all measures can be found online[49].

(4) ***Knowledge Enrichment***: This optional step, allows for the integration of the assessment results at both the individual attribute, sub-factor and overall assessment level. Our ontological representation enables us to seamlessly integrate these assessment results in the knowledge base, therefore not only supporting reuse of analysis results but also allowing their use for further semantic analysis.

7.4 Case Study

In what follows, we illustrate the applicability of our modeling approach to support the assessment of trust within OSS software libraries, by highlighting the flexibility of our modeling approach, in terms of its seamless knowledge and analysis results integration, as well as the use of Semantic Web reasoning services to infer new knowledge (measures). In Section 7.4.1, we present the setup for our study, including the selection process for the 4 projects used to illustrate our approach; Sections 7.4.2 to 7.4.4, describe how we measure security vulnerabilities, license violations, and API breaking changes. Section 7.4.5 describes how these individual identified measures can be integrated for a holistic trustworthiness assessment.

7.4.1 Study Setup

For the data collection and extraction in our case study (see Figure 7.12), we rely on four data sources: the NVD database, GitHub, SVN, and the Maven build repository. We downloaded the latest versions of the Maven and NVD repositories – which includes 1,219,731 project releases in Maven and 74,945 vulnerabilities affecting 109,212 releases in NVD. For our study,

[49] https://github.com/segps/segps-code

we limited the assessment scope to 4 projects. The projects were selected based on the following criteria: a.) at least some of their releases contained known vulnerabilities, b.) license details were provided, c.) releases varied in their major version numbers, and d.) the functionalities these products provide are widely reused by other projects (see Table 7.3 for details). The four subject systems vary in size (classes and methods) and application domain. Commons Fileupload[50] adds file upload capabilities to web applications, CXF WS Security[51] provides reusable components for client-side authentication, security, and encryption. Struts[52] is an open source framework for creating Java web applications, and ASM[53] is a Java bytecode manipulation library. We further extract the complete source code and history information of these four projects. The extracted facts are then populated in their corresponding ontologies and made persistent in our triple store.

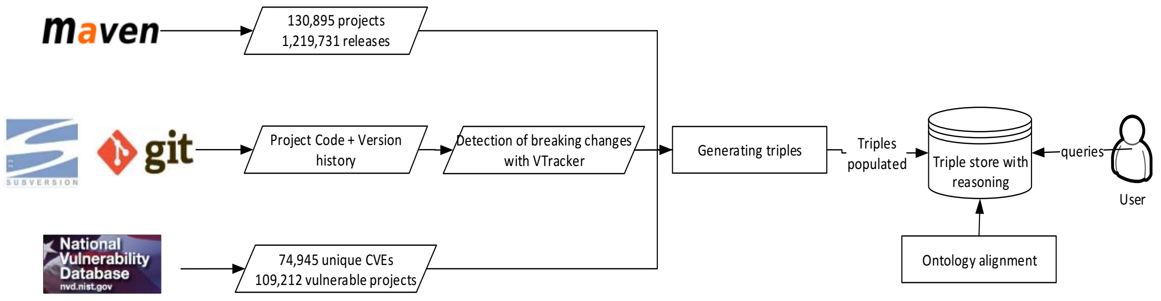

maven
130,895 projects
1,219,731 releases
SUBVERSION
git
Project Code + Version history
Detection of breaking changes with VTracker
National Vulnerability Database nvd.nist.gov
74,945 unique CVEs
109,212 vulnerable projects
Generating triples
Triples populated
Triple store with reasoning
queries
User
Ontology alignment

Figure 7.12: Overview of case study setup process

Table 7.3: Overview of selected case study projects

Project	# Releases analyzed	# of Dependencies
Commons Fileupload	6	68854
Apache CXF WS Security	5	4570
Struts	3	3170
ASM	20	8109

[50] https://commons.apache.org/proper/commons-fileupload/
[51] http://cxf.apache.org/docs/ws-security.html
[52] https://struts.apache.org/
[53] http://asm.ow2.org/

7.4.2 Identifying and Measuring Software Security Vulnerabilities

Approach. This section introduces some of the rules and queries we used to derive the WVD measures (overall, direct, and inherited). These rules are of interest, since they highlight the flexibility and power of our modeling approach, allowing users to define and customize their own derived measures without the need for any additional proprietary algorithm implementations or modeling.

WVDdirect inference: In order to derive the WVDdirect score for the projects, we define rules using the Semantic Web Rule Language (SWRL), similar to the one shown in Figure 7.13. The rule states that, if a project release has a LOC and OverallSeverityScore measure, then the release has a WVDdirect score obtained by dividing the overall severity score by LOC.

```
Release(?r),
hasLOC(?r, ?loc),
hasOverallSeverityScore(?r, ?score),
divide(?wvdDirect, ?score, ?loc)
→ hasDirectWVD(?r, ?wvdDirect)
```

Figure 7.13: Rules to infer the direct WVD measure

WVDinherit inference: For us to be able to infer the WVDinherit measure of a project release, we had first to determine the ratio of vulnerable APIs that are reused in a particular release. The OntTAM knowledge model not only captures the required information to derive this measure, but also includes all semantics to be able to take advantage of the SW reasoners to infer the measure value. More specifically, once the required ontologies (e.g., SEVONT, SEON, OntTAM) are populated, a SPARQL query can be created to retrieve the number of vulnerable API elements in a given release (see Figure 7.14).

Using Figure 7.15, we can also determine the number of such vulnerable API elements being reused in client applications. The SPARQL query (Figure 7.16) exemplifies how we take advantage of analysis results from the inference rules in Figure 7.13 to infer the final WVDinherit measure for a particular release of a component. For a more detailed description, on how we detect vulnerable code elements, the reader is referred to Chapter 6.

```
CONSTRUCT{?release sevont:hasVulnerableCodeCount ?totalVulnerableCodeCount}
WHERE{
  {
    SELECT ?release count(?vulnerableCode) as ?totalVulnerableCodeCount
    WHERE{
      ?vulnerableCode rdf:type code:VulnerableCode.
      ?release code:containsCodeEntity ?vulnerableCode
    }GROUP BY ?release
  }
}
```

Figure 7.14: SPARQL query for inferring the total number of vulnerable code entities in a project

```
CONSTRUCT{?link sevont:hasReusedVulnerableCodeCount ?usedVulnerableCodeCount}
WHERE{
{
 SELECT ?link count(?vulnerableCode) as ?usedVulnerableCodeCount
 WHERE {
  ?link a sbson:DependencyLink.
  ?link sbson:hasDependencySource ?client; sbson:hasDependencyTarget ?release.
  ?client code:containsCodeEntity ?codeEntity.
  ?codeEntity main:dependsOn ?vulnerableCode.
  {
    SELECT ?vulnerableCode
    WHERE {
      ?vulnerableCode rdf:type code:VulnerableCode.
      ?release code:containsCodeEntity ?vulnerableCode.
     }
  }
 }GROUP BY ?link
}}
```

Figure 7.15: SPARQL query for inferring the vulnerable code entities used by different dependent projects

```
CONSTRUCT{?client sevont:hasInheritWVD ?inheritWVD }
WHERE{
{
 SELECT ?client count(?indirectWVD) as ?inheritWVD
 WHERE {
  ?link a sbson:DependencyLink.
  ?link sbson:hasDependencySource ?client; sbson:hasDependencyTarget ?release.
  ?client sevont:hasReusedVulnerableCodeCount ?usedVulnerableCodeCount.
  ?release sevont:hasVulnerableCodeCount ?totalVulnerableCodeCount.
  ?release sevont:hasDirectWVD ?directWVD.
  BIND((?usedVulnerableCodeCount/?totalVulnerableCodeCount) AS ?vulnerableCodeRatio).
  BIND((?vulnerableCodeRatio * ?directWVD) AS ?indirectWVD).
 }
}}
```
Figure 7.16: SPARQL query for inferring inherited WVD measures in clients' projects

Findings and Discussion. Table 7.4 provides an overview of results from our case study, including the number of known vulnerabilities, project size, and WVD scores for selected project releases. Using the WVD measure we can now compare two releases of the same project in terms of their weighted vulnerability density. For example, based on the WVD measure, we can consider Struts 1.2.9 to be more trustworthy than earlier versions of Struts (e.g., version 1.2.4 and 1.2.8, which have both higher WVD scores). However, the latest version is not always better than earlier versions as seen with the analyzed Apache CXF WS Security libraries. Version 2.7.0 of the CXF WS Security library has a worse WVD compared to its previous versions – two new vulnerabilities were introduced in version 2.7.0 in addition to the existing vulnerabilities inherited from prior versions.

We further analyzed the WVD results, to see whether developers migrate their applications to library versions which are less vulnerable (e.g., a newer version of the same library with patched vulnerabilities). Table 7.5 provides an overview of the number of dependent applications which change their build dependency to a more trustworthy release (based on the lower WVD score). Our analysis results show that 45.1% client applications which switched their library dependencies; out of these, 63.29% switched to a more trustworthy library release. Surprisingly, the remaining 36.71% switched to library releases which are either equal or less trustworthy (higher WVD score), even if more trustworthy library versions are available.

Table 7.4: Vulnerability densities of selected projects

Project	# vulnerabilities	Aggregated Vuln. Scores	Size (Kloc)	WVD
commons-fileupload 1.0	2	10.8	1.23	8.78
commons-fileupload 1.1	2	10.8	1.28	8.46
commons-fileupload 1.2	2	10.8	1.78	6.05
commons-fileupload 1.2.1	2	10.8	1.97	5.49
commons-fileupload 1.2.2	2	10.8	2.04	5.31
commons-fileupload 1.3	1	7.5	2.39	3.14
Apache CXF WS Security 2.4.1	4	23.6	18.92	1.25
Apache CXF WS Security 2.4.4	4	23.6	21.30	1.11
Apache CXF WS Security 2.4.6	5	27.9	23.10	1.21
Apache CXF WS Security 2.6.3	8	39.4	26.43	1.49
Apache CXF WS Security 2.7.0	10	49.4	26.43	1.87
Struts 1.2.4	5	30	24.04	1.25
Struts 1.2.8	8	49.6	24.61	2.02
Struts 1.2.9	4	25.7	24.76	1.04

Table 7.5: Clients who switched from a vulnerable API in later release

Project	Vulnerability	% clients switched versions of the library	% clients switch to less vulnerable release (WVD)	% clients switch to a release with equal or higher WVD score
commons-fileupload 1.0	CVE-2014-0050	29.36%	74.26%	25.74%
commons-fileupload 1.1		6.28%	58.33%	41.67%
commons-fileupload 1.2		70.54%	100.00%	0.00%
commons-fileupload 1.2.1		38.97%	97.55%	2.45%
commons-fileupload 1.2.2		46.79%	99.99%	0.01%
commons-fileupload 1.3		40.62%	0.00%	100.00%
Apache CXF WS Security 2.4.1	CVE-2013-0239	94.93%	100.00%	0.00%
Apache CXF WS Security 2.4.4		95.00%	0.23%	99.77%
Apache CXF WS Security 2.4.6		95.24%	63.10%	36.90%
Apache CXF WS Security 2.6.3		98.08%	85.29%	14.71%
Apache CXF WS Security 2.7.0		92.75%	97.26%	2.74%
Struts 1.2.4	CVE-2016-1181	0.00%	n/a	n/a
Struts 1.2.8		44.44%	100.00%	0.00%
Struts 1.2.9		0.00%	n/a	n/a

7.4.3 Identifying and Measuring License Violations

Approach. License violations originating from external libraries and components can cause a major long-term liability for client applications, which can have a negative effect on the use of third-party intellectual property and therefore their trustworthiness of these libraries. In our study, we first evaluate if such license violations (non-compliances) occur in general in project dependencies managed by the Maven repository. In the second part of our study, we revisit our 4 projects used in our trustworthiness assessment study, to assess their trustworthiness in terms of license violations. For the study, we create SPARQL queries that analyze all dependency relationships in the Maven repository and identify three main categories of license violations: simple violations, transitive violations, and compound violations (see Section 7.3.3). The queries take advantage of both our open source license ontology and the build ontology. Figures 7.17, 7.18, and 7.19 below illustrates the queries we used to identify these violations.

```
SELECT distinct *
WHERE {
 ?link a sbson:DependencyLink.
  ?link sbson:hasDependencyTarget ?project2.
  ?link sbson:hasDependencySource ?project1.
  ?project1 markosLicense:coveringLicense ?license1.
  ?project2 markosLicense:coveringLicense ?license2.
  ?license1 markosCopyright:incompatibleWith ?license2.
}
```

Figure 7.17: SPARQL query for inferring the total number of simple license violations

```
SELECT distinct *
WHERE {
 ?linkA a sbson:DependencyLink.
  ?linkA sbson:hasDependencyTarget ?project2.
  ?linkA sbson:hasDependencySource ?project1.
  ?linkB a sbson:DependencyLink.
  ?linkB sbson:hasDependencyTarget ?project3.
  ?linkB sbson:hasDependencySource ?project2.
  ?project1 markosLicense:coveringLicense ?license1.
  ?project2 markosLicense:coveringLicense ?license2.
  ?project3 markosLicense:coveringLicense ?license3.
  ?license1 markosCopyright:compatibleWith ?license2.
  ?license2 markosCopyright:compatibleWith ?license3.
  ?license1 markosCopyright:incompatibleWith ?license3.
}
```

Figure 7.18: SPARQL query for inferring the total number of transitive license violations

```
SELECT distinct *
WHERE {
 ?linkA a sbson:DependencyLink.
  ?linkA sbson:hasDependencyTarget ?project2.
  ?linkA sbson:hasDependencySource ?project1.
  ?linkB a sbson:DependencyLink.
  ?linkB sbson:hasDependencyTarget ?project3.
  ?linkB sbson:hasDependencySource ?project1.
  ?project1 markosLicense:coveringLicense ?license1.
  ?project2 markosLicense:coveringLicense ?license2.
  ?project3 markosLicense:coveringLicense ?license3.
  ?license1 markosCopyright:compatibleWith ?license2.
  ?license1 markosCopyright:compatibleWith ?license3.
  ?license2 markosCopyright:incompatibleWith ?license3.
}
```

Figure 7.19: SPARQL query for inferring the total number of compound license violations

Findings and Discussion. This section presents and discusses the results which we obtained from our license violation experiment using the Maven repository. Figure 7.20 shows the distribution of different project licenses across the Maven repository. In Table 7.6, we report on the license violations (classified by the type of violation), which we identified in our study of the Maven repository.

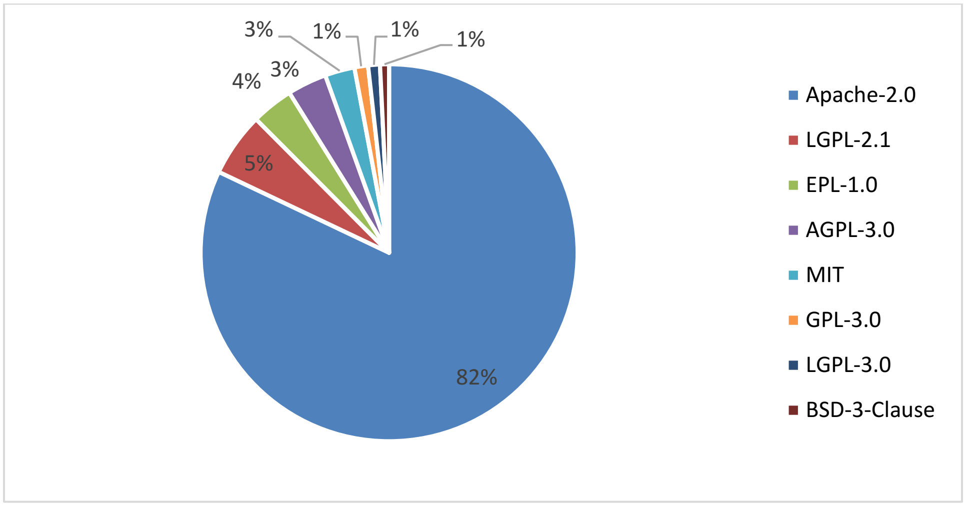


Figure 7.20: License distribution in the Maven repository

Table 7.6: Totals for each type of violation found by querying the data store

License Violation Types	Count
Type 1 - Simple Violations	131996
Type 2 - Transitive Violations	288153
Type 3 - Compound Violations	654964

Our study identified over 131,000 simple violations and numerous transitive license violations of different types. We note that Type 3 violations are seemingly the most popular type of violation, followed by Type 2, then 1. In what follows, we discuss in more detail some of the license violations or incompatibilities which we observed in our study.

Figures 7.21, 7.22, and 7.23 summarize the most common license violation pairs which occurred for all three license violation categories. The most common Type 1 violation which we observed is code published under the Apache 2 license being incorporated into GPL 2 licensed code. This violation is not surprising for two reasons. First, many software developers are simply not aware nor well-versed in open source license compliance, and as these are the two of the most popular licenses in the world, this pairing reflects their usage in the wild. Second, there is likely some confusion about Apache 2's compatibility with the GPL. On the GNU website, the Free Software Foundation publishes a list of licenses that are compatible with the GPL. This page shows Apache 2 in green (meaning compatible), but in the license discussion, the authors explain that Apache 2 is only compatible with GPL 3, not GPL 2 [146].

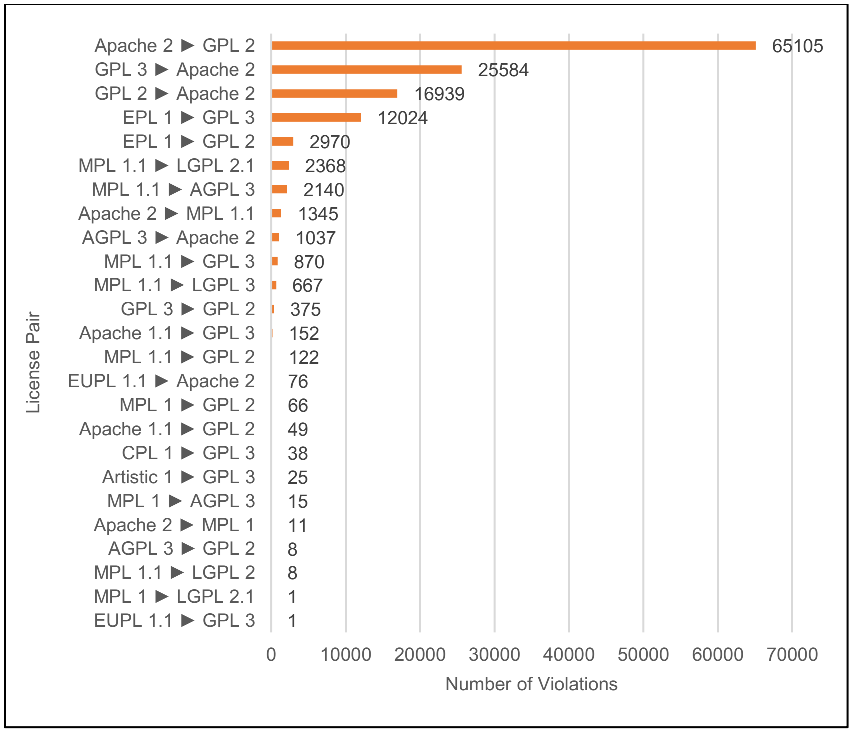


Figure 7.21: Most Popular Type 1 License Violation Pairs

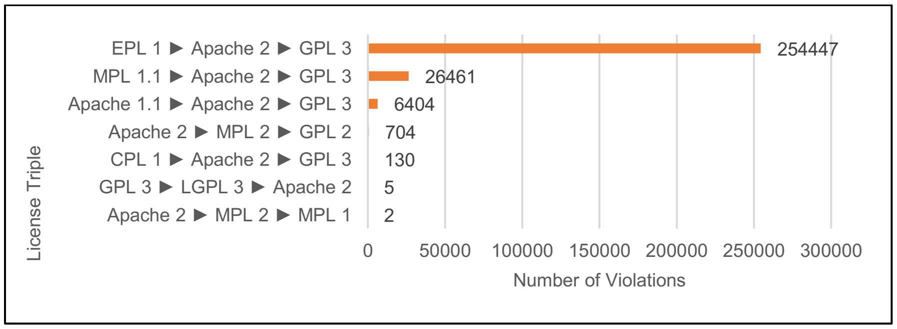


Figure 7.22: Most Popular Type 2 License Violation Pairs

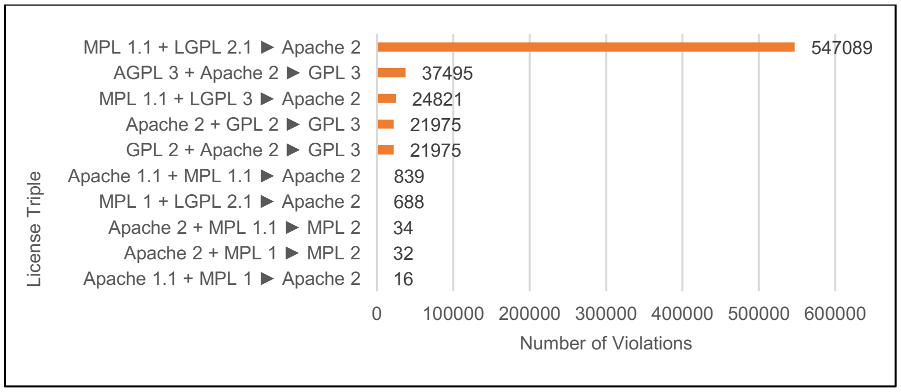


Figure 7.23: Most Popular Type 3 License Violation Pairs

A more detailed analysis of the reasons why the number of transitive license violations is significantly larger compared to direct violations revealed: (1) Type 1 license compatibility/incompatibility are easier to verify/detect. That is, it is much more likely that a developer will check for license compliance when only two licenses are involved. (2) Transitive violation types, on the other hand, have not been considered in the research community prior to this work, and may very well be acceptable or be clearly identifiable as such. For example, the European Union Public License (EUPL) explicitly states which licenses it is compatible with. This is a known compatibility. Whereas for transitive interactions, the EUPL may then be imported into an intermediary project, say a project under the Licence Libre du Québec – Réciprocité (LiLiQ-R), which is then imported into a tertiary project under Common Development and Distribution License (CDDL). Each step (EUPL to LiLiQ, and LiLiQ to CDDL) is known to be compatible. But the EUPL does not explicitly state that it is compatible with the CDDL. This chain of licenses may be flagged as a violation by our approach. Yet this chain could, in fact, be perfectly lawful (a false-positive, verifiable by a lawyer). Our approach will, however, flag such a dependency chain as a potential violation. This triple is neither a known compatibility nor known incompatibility and thus is one of the reasons why there are more Type 2 violations found.

Identification of Type 3 violations becomes even more difficult to detect since their detection largely depends on how licenses define derivative works and conditions for reusing these libraries. Libraries can be used by either including the actual source code or through linking (e.g. through a jar file). Linking of a library can be static (compile-time) or dynamic (run-time). For example, LGPL requires each project to be an "*independent work that stands by itself and includes no source code from [the other]*." In this scenario, it is perfectly acceptable to combine the compiled code, however [170]. So basically, the question is whether a derivative work is created or not, when combining dependencies into a new project. Derivative works come into play only when the licensed software is copied, distributed, or modified. Additional research is needed to further clarify legal and license compliance issue when using these open source licenses. However, as can be noted, all three types of violations can exist in projects. Thus, simple, transitive, and complex license violations are problems that occur in open source projects and can potentially affect the trustworthiness of components and libraries being reused in software projects.

In what follows, we report on license violations results which we observed for the selected 4 projects in our trustworthy study. Table 7.7 provides an overview of the number of license violations which we detected in these projects. Only four (4) releases of Commons-Fileupload introduced violations in client applications. For the remaining projects, no license violations are reported due to the lack of license information in the analyzed client applications. Results, although incomplete, confirm our previous claim that violations are problems that occur in open source projects.

Table 7.7: Licence Violation Counts in selected projects

Project	# Simple Violations	# Transitive Violations	# Compound Violations
commons-fileupload 1.0	0	0	0
commons-fileupload 1.1	0	0	0
commons-fileupload 1.2	4	0	0
commons-fileupload 1.2.1	14	0	0
commons-fileupload 1.2.2	19	0	0
commons-fileupload 1.3	4	0	0
Apache CXF WS Security 2.4.1	0	0	0
Apache CXF WS Security 2.4.4	0	0	0
Apache CXF WS Security 2.4.6	0	0	0
Apache CXF WS Security 2.6.3	0	0	0
Apache CXF WS Security 2.7.0	0	0	0
Struts 1.2.4	0	0	0
Struts 1.2.8	0	0	0
Struts 1.2.9	0	0	0

7.4.4 Identifying and Measuring API Breaking Changes

Approach. As previously mentioned in our study setup (Section 7.4.1, Figure 7.12), we extract the source code and versioning information of the four projects from GitHub and SVN. We identify the introduced breaking and non-breaking changes for each successive pair of releases of a given project using the VTracker tool. In order to be able to reuse the analysis results for further analysis, we take advantage of our ontological knowledge modeling approach and extend our knowledge base to include the analysis results. Developers can now access this information, using SPARQL queries, to derive potential direct and indirect impacts of breaking changes on their client applications. Complete details on how we identify, and model breaking changes can be found in Chapter 5. In what follows, we show some of the main rules and queries used to derive the BCD and BCI measures.

BCD inference: For computing the BCD scores of the projects in our dataset, we define a SWRL rule (see Figure 7.24), which infers the BCD score from the breaking and non-breaking change counts. Figures 7.25 and 7.26 detail the queries for computing the breaking and non-breaking change measures of a project.

BCI_{direct} and $BCI_{indirect}$ inference: The queries in Figures 7.27 and 7.28 take advantage of the inference services to derive both the direct and indirect BCI scores from a project and its dependencies. The query in Figure 7.28 first identifies two unique releases of the same project for which breaking changes have been populated into the triple-store. It then identifies any usage of the found binary incompatible APIs within the client. These queries are based on Equations 6 and 7 in Section 7.3.3.

```
Release(?r),
hasBreakingChangeCount(?r, ?bcc),
hasNonBreakingChangeCount (?r, ?nbcc),
divide(?bcd, ?bcc, ?nbcc)
→ hasBCD(?r, ?bcd)
```

Figure 7.24: SWRL rules to infer the BCD measure

```
CONSTRUCT{?release code:hasBreakingChangeCount ?totalBreakingChanges }
WHERE{
{
 SELECT ?release count(?breakingChange) as ?totalBreakingChanges
 WHERE{
  ?breakingChange rdf:type code:BreakingChange.
  ?breakingChange code:hasCurrentAPI ?api.
  ?release code:containsCodeEntity ?api.
 }GROUP BY ?release
}}
```

Figure 7.25: SPARQL query for inferring the total number of breaking changes in a project

```
CONSTRUCT{?release code:hasNonBreakingChangeCount ?totalNonBreakingChanges }
WHERE{
{
 SELECT ?release count(?nonbreakingChange) as ?totalNonBreakingChanges
 WHERE{
  ?nonbreakingChange rdf:type code:NonBreakingChange.
  ?nonbreakingChange code:hasCurrentAPI ?api.
  ?release code:containsCodeEntity ?api.
 }GROUP BY ?release
}}
```

Figure 7.26: SPARQL query for inferring the total number of non-breaking changes in a project

```
CONSTRUCT{?release code:hasDirectBCI ?directBCI }
WHERE{
{
 SELECT ?release ?directBCI
 WHERE {
  BIND((?usedBreakingChanges/?bcc) AS ?directBCI).
  {
   SELECT ?release count(?breakingApi) as ?usedBreakingChanges ?bcc
   WHERE{
    ?breakingChange rdf:type code:BreakingChange; code:hasCurrentAPI ?breakingApi;
    ?dependent code:containsCodeEntity ?breakingApi; code:hasBreakingChangeCount ?bcc.
    ?client code:containsCodeEntity ?api.
    ?api main:dependsOn ?breakingApi.
   }GROUP BY ?release
  }
 }
}}
```

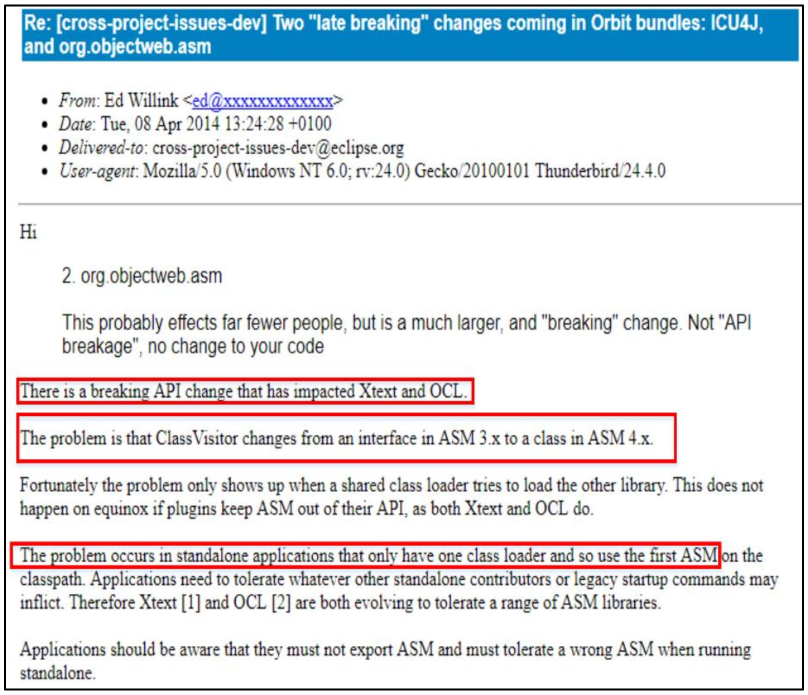
Figure 7.27: SPARQL query for inferring the direct BCI measure in a project

```
CONSTRUCT{?client  code:hasIndirectBCI ?indirectBCI }
WHERE{
{
 SELECT ?client ?indirectBCI
 WHERE {
  BIND((?usedBreakingChanges/?bcc) AS ?indirectBCI).
  {
   SELECT ?client count(?clientAPIEntity) as ?usedBreakingChanges count(?breakingChange) as ?bcc
   WHERE{
    #identify use of breaking change entity in clien
    ?client code:containsCodeEntity ?clientAPIEntity.
    {?clientAPIEntity main:dependsOn ?currentAPIElement} UNION
    {?clientAPIEntity main:dependsOn ?priorAPIElement}.
    {
     SELECT ?client, ?dependency ?asm1, ?asm2
     WHERE {
      #Identify different releases of the same project for which breaking changes exist
      ?client sbson:hasBuildDependencyOn ?dependency1; sbson:hasBuildDependencyOn ?dependency2.
      ?breakingChange a code:BreakingCodeChange.
      ?breakingChange code:hasPriorAPI ?priorAPIElement; code:hasCurrentAPI ?currentAPIElement.
      ?dependency1 code:containsCodeEntity ?currentAPIElement.
      ?dependency2 code:containsCodeEntity ?priorAPIElement.
      FILTER(?dependency1 != ?dependency2).
     }
    }GROUP BY ?client
   }
  }
}}
```

Figure 7.28: SPARQL query for inferring the indirect BCI measure in a project

Findings and Discussion. Figure 7.29 shows an example of a bug[54] reported in Eclipse Orbit[55]. Orbit depends on ASM, a Java bytecode manipulation library. ASM introduced breaking changes in its later releases, such as ClassVisitor being changed from an interface (version 3.X) to a class in version 4.0. This change is a major change in the API and therefore breaking the older 3.X API releases.

[54] https://dev.eclipse.org/mhonarc/lists/cross-project-issues-dev/msg10487.html
[55] https://www.eclipse.org/orbit/


Re: [cross-project-issues-dev] Two "late breaking" changes coming in Orbit bundles: ICU4J, and org.objectweb.asm

- *From*: Ed Willink <ed@xxxxxxxxxxxxx>
- *Date*: Tue, 08 Apr 2014 13:24:28 +0100
- *Delivered-to*: cross-project-issues-dev@eclipse.org
- *User-agent*: Mozilla/5.0 (Windows NT 6.0; rv:24.0) Gecko/20100101 Thunderbird/24.4.0

Hi

> 2. org.objectweb.asm
>
> This probably effects far fewer people, but is a much larger, and "breaking" change. Not "API breakage", no change to your code

There is a breaking API change that has impacted Xtext and OCL.

The problem is that ClassVisitor changes from an interface in ASM 3.x to a class in ASM 4.x.

Fortunately the problem only shows up when a shared class loader tries to load the other library. This does not happen on equinox if plugins keep ASM out of their API, as both Xtext and OCL do.

The problem occurs in standalone applications that only have one class loader and so use the first ASM on the classpath. Applications need to tolerate whatever other standalone contributors or legacy startup commands may inflict. Therefore Xtext [1] and OCL [2] are both evolving to tolerate a range of ASM libraries.

Applications should be aware that they must not export ASM and must tolerate a wrong ASM when running standalone.

Figure 7.29: An example of a reported bug showing how a breaking change in the ASM library impacts Orbit and its dependent projects

We also illustrate how our ontology-based API dependency measures can aid developers in detecting and dealing with such breaking changes. For the analysis, we extract and populate facts about the breaking changes between different versions of ASM releases and the source code of all projects which directly depend on ASM releases (8109 dependencies in total). Based on the extracted source code and dependency information, the earlier introduced SPARQL queries can now be used to identify the potential direct and indirect impacts of ASM breaking changes on client applications.

Figure 7.30 shows the distribution of (a) breaking changes, (b) non- breaking changes, and (c) breaking change densities (BCD) across all selected 20 ASM releases. Figure 7.30(d) reports on the impact of the ClassVisitor API breaking change on client applications. Furthermore, this change can potentially affect, on average, 50 different API elements and as many as 225 API elements in a single client application. The reported impact set returned by our approach includes clients which reuse the ClassVisitor API either directly (through an implementation of the interface) or indirectly (through transitive inheritance or method invocations).

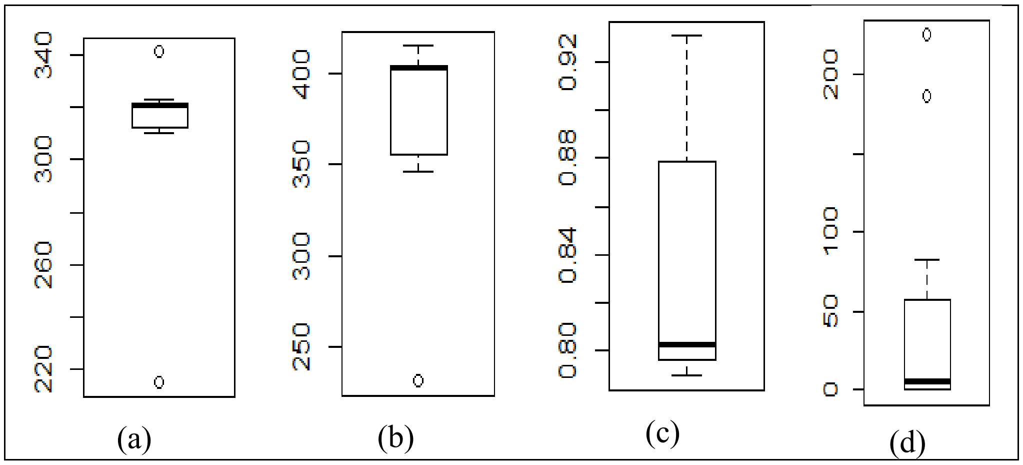


Figure 7.30: Distribution of breaking changes and their impacts in the analyzed ASM libraries and dependencies

7.4.5 Assessment Process

In the previous sub-sections we described how we identify and measure different attributes of trustworthiness by taking advantage of our unified ontological knowledge representation and SW reasoning services. The OntTAM assessment process further integrates these scores across attributes and sub-factors. For the actual assessment process, we first compute the fuzzy score for each measure individually and then aggregate these scores to calculate the attribute, sub-factors, factors, and dimension assessment scores. Figure 7.31 gives a complete overview of how the sub-factors, attributes, and measures are related and used to derive our trustworthiness assessment.

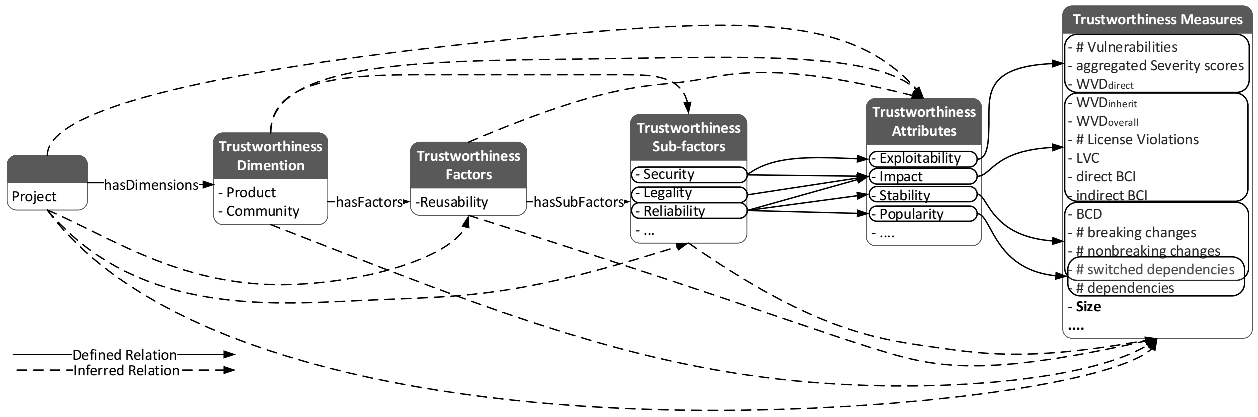


Figure 7.31: Overview of relations in the semantic OntTAM domain model

The effect of the fuzzification on the assessment scores typically increases with the assessment abstraction levels (e.g., quality dimension scores vs attribute scores). Figures 7.32 and 7.33 show the rules we used to create the fuzzified score for the WVD measure and Figure 7.34 provides example rules we used to combine the fuzzified LVC and WVD scores into a score for the Impact attribute.

```
FUNCTION_BLOCK WVD

VAR_INPUT
  WVD_Measure: REAL;
  WVD_Weight: REAL;
END_VAR

VAR_OUTPUT
  WVD_Score: REAL;
END_VAR

FUZZIFY WVD_Measure
  TERM VERYLOW := (0.0,1.0) (1.04,1.0) (2.11,0.0) ;
  TERM LOW := (1.90,0.0) (2.975,1.0) (4.14,0.0) ;
  TERM AVERAGE := (3.73,0.0) (4.91,1.0) (6.17,0.0) ;
  TERM HIGH := (5.55,0.0) (6.845,1.0) (8.20,0.0) ;
  TERM VERYHIGH := (7.38,0.0) (8.78,1.0) (11.29,1.0) ;
END_FUZZIFY

FUZZIFY WVD_Weight
  TERM LOW := (0.0,1.0) (0.5,1.0) (2.69,0.0) ;
  TERM MEDIUM := (2.56,0.0) (4.75,1.0) (7.05,0.0) ;
  TERM HIGH := (6.69,0.0) (9.0,1.0) (12.0,1.0) ;
END_FUZZIFY

DEFUZZIFY WVD_Score
  TERM VERYPOOR := (6.5,0.0) (7.5,1.0) (9.0,1.0) ;
  TERM POOR := (5.31,0.0) (6.25,1.0) (7.22,0.0) ;
  TERM AVERAGE := (4.14,0.0) (5.0,1.0) (5.9,0.0) ;
  TERM VERYGOOD := (2.95,0.0) (3.75,1.0) (4.6,0.0) ;
  TERM EXCELLENT := (0.0,1.0) (2.5,1.0) (3.28,0.0) ;

  METHOD : COG;
END_DEFUZZIFY
```

Figure 7.32: Sample FCL file for defining the fuzzy WVD measure

```
RULEBLOCK WVD_SCORE_RULES
RULE 0 : IF WVD_Measure IS VERYLOW AND WVD_Weight IS LOW      THEN   WVD_Score IS
EXCELLENT ;
RULE 1 : IF WVD_Measure IS VERYLOW AND WVD_Weight IS MEDIUM THEN WVD_Score IS
EXCELLENT ;
RULE 2 : IF WVD_Measure IS VERYLOW AND WVD_Weight IS HIGH THEN WVD_Score IS
VERYGOOD ;
RULE 3 : IF WVD_Measure  IS LOW AND WVD_Weight IS LOW THEN WVD_Score IS EXCELLENT ;
RULE 4 : IF WVD_Measure  IS LOW AND WVD_Weight IS MEDIUM THEN WVD_Score IS
VERYGOOD ;
RULE 5 : IF WVD_Measure  IS LOW AND WVD_Weight IS HIGH THEN WVD_Score IS AVERAGE ;
RULE 6 : IF WVD_Measure  IS AVERAGE AND WVD_Weight IS LOW THEN WVD_Score IS
VERYGOOD ;
RULE 7 : IF WVD_Measure  IS AVERAGE AND WVD_Weight IS MEDIUM THEN WVD_Score IS
AVERAGE ;
RULE 8 : IF WVD_Measure  IS AVERAGE AND WVD_Weight IS HIGH THEN WVD_Score IS POOR;
RULE 9 : IF WVD_Measure  IS HIGH AND WVD_Weight IS LOW THEN WVD_Score IS AVERAGE ;
RULE 10 : IF WVD_Measure  IS HIGH AND WVD_Weight IS MEDIUM THEN WVD_Score IS POOR ;
RULE 11 : IF WVD_Measure  IS HIGH AND WVD_Weight IS HIGH THEN WVD_Score IS
VERYPOOR;
RULE 12 : IF WVD_Measure  IS VERYHIGH AND WVD_Weight IS LOW THEN WVD_Score IS POOR ;
RULE 13 : IF WVD_Measure  IS VERYHIGH AND WVD_Weight IS MEDIUM THEN WVD_Score IS
VERYPOOR ;
RULE 14 : IF WVD_Measure  IS VERYHIGH AND WVD_Weight IS HIGH THEN WVD_Score IS
VERYPOOR ;
END_RULEBLOCK
END_FUNCTION_BLOCK
```

Figure 7.33: Sample FCL file for inferring the fuzzy scores for the WVD measure

Using the property chain axioms, which we explained earlier in Section 7.3.2.1, one can now automatically infer trustworthiness scores from the populated measures of any given project. Figure 7.35 provides a list of sample queries used for integration and fuzzification.

```
RULEBLOCK IMPACT _SCORE_RULES

RULE 0 : IF LVC_Score IS EXCELLENT AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS
AVERAGE ;

RULE 1 : IF LVC_Score IS VERYGOOD AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS
POOR ;

RULE 2 : IF LVC_Score IS AVERAGE AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS
POOR ;

RULE 3 : IF LVC_Score IS POOR AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS
VERYPOOR ;

RULE 4 : IF LVC_Score IS VERYPOOR AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS
VERYPOOR;

…

END_RULEBLOCK

END_FUNCTION_BLOCK
```

Figure 7.34: Sample FCL file for integrating the LVC and WVD fuzzy scores for the Impact attribute

Query 1: At sub-factor level

```
SELECT distinct ?project ?subfactorScore
WHERE {
 ?impactAttribute a onttam:SubFactor.
 ?project onttam:hasSubfactor ?subfactorAttribute.
 ?subfactorAttribute onttam:hasScore  ?subfactorScore.
}
```

Query 2: At factor level

```
SELECT distinct ?project ?factorScore
WHERE {
 ?factorAttribute a onttam:Factor.
 ?project onttam:hasFactor ?factorAttribute.
 ?factorAttribute onttam:hasScore  ?factorScore._
}
```

Figure 7.35: SPARQL query illustrating the inference of overall trustworthiness scores

Findings and Discussion. Table 7.8 presents a summary of trustworthiness scores, which we derived from the three software trustworthiness categories we consider in the scope of this work: API breaking changes, security vulnerabilities, and license violations.

Table 7.8: Overview of selected trustworthiness measure scores for our case study projects

Project	WVD		LVC		BCD	
	Numeric Score	Fuzzified Score	Numeric Score	Fuzzified Score	Numeric Score	Fuzzified Score
commons-fileupload 1.0	8.78	VERYPOOR	0	EXCELLENT	0	EXCELLENT
commons-fileupload 1.1	8.46	VERYPOOR	0	EXCELLENT	2.14	VERYPOOR
commons-fileupload 1.2	6.05	POOR	4	VERYPOOR	0.64	POOR
commons-fileupload 1.2.1	5.49	AVERAGE	14	VERYPOOR	0.49	AVERAGE
commons-fileupload 1.2.2	5.31	AVERAGE	19	VERYPOOR	0.48	AVERAGE
commons-fileupload 1.3	3.14	VERYGOOD	4	VERYPOOR	0.6	AVERAGE
Apache CXF WS Security 2.4.1	1.25	EXCELLENT	0	EXCELLENT	0.08	EXCELLENT
Apache CXF WS Security 2.4.4	1.11	EXCELLENT	0	EXCELLENT	0.95	VERYPOOR
Apache CXF WS Security 2.4.6	1.21	EXCELLENT	0	EXCELLENT	0.89	VERYPOOR
Apache CXF WS Security 2.6.3	1.49	EXCELLENT	0	EXCELLENT	0.86	VERYPOOR
Apache CXF WS Security 2.7.0	1.87	EXCELLENT	0	EXCELLENT	0.88	VERYPOOR
Struts 1.2.4	1.25	EXCELLENT	0	EXCELLENT	0.9	VERYPOOR
Struts 1.2.8	2.02	EXCELLENT	0	EXCELLENT	0.44	AVERAGE
Struts 1.2.9	1.04	EXCELLENT	0	EXCELLENT	0.32	VERYGOOD

Table 7.9: Example of inferred trustworthiness scores at sub-factor level

Project	Security SubFactor		Legality SubFactor		Reliability SubFactor	
	Numeric Score	Fuzzified Score	Numeric Score	Fuzzified Score	Numeric Score	Fuzzified Score
commons-fileupload 1.0	5.01	AVERAGE	1.45	EXCELLENT	0	EXCELLENT
commons-fileupload 1.1	5.01	AVERAGE	1.45	EXCELLENT	0	EXCELLENT
commons-fileupload 1.2	5.01	AVERAGE	1.45	EXCELLENT	0	EXCELLENT
commons-fileupload 1.2.1	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
commons-fileupload 1.2.2	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
commons-fileupload 1.3	3.77	VERYGOOD	1.45	EXCELLENT	0	EXCELLENT
Apache CXF WS Security 2.4.1	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
Apache CXF WS Security 2.4.4	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
Apache CXF WS Security 2.4.6	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
Apache CXF WS Security 2.6.3	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
Apache CXF WS Security 2.7.0	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
Struts 1.2.4	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
Struts 1.2.8	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT
Struts 1.2.9	1.45	EXCELLENT	1.45	EXCELLENT	0	EXCELLENT

Tables 7.9 and 7.10 report on the results from our queries in Figure 7.35. The results indicate that despite the presence of security, licensing, and breaking change concerns, almost all projects have excellent trustworthiness scores at the presented sub-factor and factor levels. This is due to

how the score categories are distributed over the fuzzy scale. In our work, the categories are distributed equally from 0 to maximum measure value recorded in our dataset. For example, the maximum WVD measure in our dataset is 11.29, making all WVD measures under 2.95 excellent. The complete scale distributions for all our measures can be found in the FCL files online[56].

Table 7.10: Example of inferred trustworthiness scores at factor level

Project	Reusability Factor	
	Numeric Score	Fuzzified Score
commons-fileupload 1.0	0	EXCELLENT
commons-fileupload 1.1	0	EXCELLENT
commons-fileupload 1.2	0	EXCELLENT
commons-fileupload 1.2.1	1.45	EXCELLENT
commons-fileupload 1.2.2	1.45	EXCELLENT
commons-fileupload 1.3	1.45	EXCELLENT
Apache CXF WS Security 2.4.1	1.45	EXCELLENT
Apache CXF WS Security 2.4.4	1.45	EXCELLENT
Apache CXF WS Security 2.4.6	1.45	EXCELLENT
Apache CXF WS Security 2.6.3	1.45	EXCELLENT
Apache CXF WS Security 2.7.0	1.45	EXCELLENT
Struts 1.2.4	1.45	EXCELLENT
Struts 1.2.8	1.45	EXCELLENT
Struts 1.2.9	1.45	EXCELLENT

It should be noted that the tables above do not report on the final overall trustworthiness score since this score would require a particular assessment context and an instantiation of our OntTAM assessment model with more measures, attributes, and sub-factors.

7.5 Discussion and Related Work

7.5.1 Threats to Validity

7.5.1.1 Internal Threats

A potential threat to our approach is whether the set of measures we considered in our assessment as part of OntTAM evaluation is sufficient to capture reusability as a trustworthiness factor. We addressed this threat by selection our trustworthiness measures from a well-

[56] https://github.com/segps/segps-code/tree/master/segps.onttam/src/main/resources/segps/onttam/fcl/measures

established subset of existing trustworthiness models, such as PAS 754:2014, QualiPSo [171], and Boland et. al. [172]. While we only selected a very small subset of these trustworthiness attributes, we believe this subset is sufficient to illustrate the applicability of our assessment model. In particular, the objective of our study was not to verify the assessment model for its completeness but rather to illustrate that OntTAM can be instantiated to a given (user specified) assessment context. The study shows that instantiating and extending OntTAM to support other requirements including new measures, attributes or sub-factors is a straightforward task.

7.5.1.2 External Threats

Definition of license violations and compliance. Given the large number of licenses available in the open source community, there exists currently no comprehensive conceptual framework describing the dependencies among all these licenses. There is a need for involving both the development community and intellectual copyright experts to consolidate and redefine the dependencies among the various open source licenses. The objective of our work is to formalize and conceptualize license violations as a domain of discourse at the TBox level. Actual license dependencies can be inferred once the ontology is populated (ABox) with available license dependency information, therefore allowing us to take advantage of ontologies and their ability to deal with incremental knowledge population and incomplete knowledge inference.

7.5.2 Related Work

7.5.2.1 Library Recommendation and Migration Techniques

Many third-party libraries are available for download to reduce development time by providing access to features ready for use. To help developers take advantage of these libraries, several techniques have been proposed that provide automatic library recommendations to developers. Common to these approaches is that they rely on criteria such as popularity and stability. Some of them even rely on the client's context (e.g., mining previous usage of libraries) for their recommendations. For example, Mileva et al. [133] study the popularity of an API. Their approach studies the rate at which dependencies adopt or switch from OSS libraries. Hora and Valente [143] build on Mileva's approach to introduce four distinct API popularity trends: fast growth, constant growth, peak growth, and dead growth. Their approach can benefit both library developers and clients. For example, library developers can be notified when the popularity of their API begins to go down. Raemaekers et al. [117] present four stability metrics that calculate the stability of API interfaces. They demonstrate how the metrics can be used by developers in deciding on libraries to reuse. The frequency of the migration of API dependencies has also been used to determine the stability of an API by [2], [16], [143].

Other techniques exist which recommend various API elements (method calls, blocks of code, etc.) of a software library to developers using heuristics that leverage various information sources (source code, commit logs, etc.). Thung et al. [142] propose an automated technique, which combines association rule mining techniques and collaborative filtering to perform the recommendation of libraries. Their approach recommends a number of likely relevant libraries to developers of a target project based on the libraries used by other projects. McCarey et al. recommend methods of software libraries to a developer by investigating the history of methods that have been used in the past [144].

In addition, several API documentation and tutorial analysis approaches have been introduced to aid developers understand how the features provided by software libraries can be correctly utilized (e.g., [173], [174]).

The above-mentioned techniques rarely consider the impact of reused external libraries on the quality of a client's project. Our work aims to provide developers with an approach to assess how much trust can be placed on a recommended software library. Our work can be seen complementary to existing library recommendation systems, in terms of extending these existing recommendation criteria by making quality in the form of trustworthiness an integrated part of the library recommendations.

7.5.2.2 Software License Violation Identification

Related studies into identifying software license violations can be categorized into two levels: intra-project and inter-project. At the intra-project level, studies aim to identify the introduction of license violations introduced by having project files with different licenses. Di Penta et al. [175] proposed an approach to automatically track the licensing evolution of systems, identifying changes in licenses and copyright years. They found that OSS projects do change licenses over time and these changes were not just to new versions of the existing license. Sometimes projects who switched licenses altogether had intended and unintended effects on downstream users of these projects. As recently as 2015, research has been conducted by Wu et al. [176] on the evolution of the licenses specified in the header of each file, with the explicit goal of finding license inconsistencies. They categorize the evolution of licenses as a license addition/removal, upgrade/downgrade, or change. These categorizations are then used to judge whether the new modification/evolution of the license results in an inconsistency.

Identifying license violations at the inter-project level requires substantial effort because developers typically combine APIs from different libraries to solve problems [177]. Several researchers have studied how reuse of source code (through cloning) and software components/libraries can lead to the introduction of license violations. Using code clones to detect small-scale license violations was touched upon by several researchers. Monden et al. [178] introduce three quality metrics for code clone detection. Disappointingly, the authors did not find any actual license violations in OSS. License violations were merely used as a theoretical use case for their comparative study. The Binary Analysis Tool (BAT) developed by Hemel et al. [179] detects code clones of OSS in proprietary binaries for the express purpose of finding violations of popular GPL projects. The authors used the comparison of string literals, data compression, and binary deltas. Interestingly, BAT does find many true code clones but falls short by leaving the verification as a manual process, i.e. whether a code clone is also a license violation.

The work by German and Hassan [180] is the most closely related to our work. The authors created a "model to describe licenses and the implications of licenses on the reuse of components." Their model describes what usage scenarios result in a derived work or not. Our work builds upon the existing body of knowledge for license violations, by providing the first attempt to create a formal representation of the license dependencies. Our approach considers the complexity and dependencies of real projects, where multiple licenses are often involved, to support the detection of license violations. The advantage of our ontological representation, being an integrated part of our unified knowledge model, is the ability to extend and reuse our license model for different type of analysis tasks, such as its seamless integration in a trustworthiness assessment model.

7.5.2.3 Quality Models

Assessing quality to improve the evolution of software systems has been addressed in existing research through the introduction of software quality models. These models introduced quality dimensions and classified quality factors that affect the development and maintenance of software products. Among the most widely accepted quality assessment model is the ISO 9126[57] software quality model standard which defines a quality model via a set of quality characteristics and sub-characteristics that were believed to be the more representative and relevant at the time of its introduction. As the complexity and vulnerability of software systems grows as a result of their components being increasingly reused across project boundaries and interconnected through networks and communication links, assessing the trustworthiness of systems and their components plays an ever-increasing role While security and interoperability are already present in the ISO 9126 standard as “sub-factors” of functionality, more recent quality models such as the ISO 25000 standard have extended the ISO 9126, by making security and interoperability a main quality aspect of the standard.

In [9], the authors introduced an SE- Evolvable QUality Assessment Meta-model (SE-EQUAM), a quality assessment model which is both evolvable and reusable. The model introduces a set of complementary core requirements necessary for a model to be considered an evolvable model: Model Reusability, Knowledge Modeling, Knowledge Population, and Knowledge Exploration [9]. In this work, we adopt the model evolvability criteria to derive our trustworthiness meta-model that is not only capable of dealing with continuous change (in the model) but also allows for its reuse by simplifying the instantiation of new domain model instances.

7.5.2.4 Trustworthiness Models

Existing work on assessing the trustworthiness of OSS systems, for example, Taibi et. al. [181], Larson et. al [182], and Tan et. al [183] have attempted to quantify OSS trustworthiness of software systems in situ, but results are limited to artifacts in the development environment; external and heterogeneous knowledge sources are not considered in these approaches. Other researchers Pfleeger et. al. [184], and Yang et. al. [185] seek to analyze and predict aspects of trustworthiness during software development. While other work has focused on introducing new evaluation criteria to better capture the nature of OSS’s components, for example, the QualiPSo model of OSS trustworthiness [171], and Boland et. al [172] quantify and assess risk based on the Structured Assurance Case Model (SACM) [186] to determine software trustworthiness. The main objective of these models is to apply their quality (trustworthy) factors to allow for a

[57] http://www.sqa.net/iso9126.html

standardized product comparison across different projects and domains. Most trustworthiness assessment models share a generic structure, template, or frame for assessing software security quality that corresponds to a hierarchy or tree structure with multiple levels and a set of constraints that define the relationship between one level and the next one. However, regardless of the kind of components, these syntactic proposals mainly address and mostly focus on the evaluation criteria and decision-making phases, setting aside the practical problem of how to search for and locate components and to assign suitable information about them [187]. Also, a general concern in most of these models is that they rely on the software product and traditional software lifecycle artifacts. They do not necessarily consider external resources in their assessment such as external vulnerability databases. As a result, there is no consensus on the applicability of these trustworthiness models in industrial practice [188].

While existing proposals for the creating such meta-modeling assessment models focus on adopting one or more of these existing quality models in one standard model, this may result in an incomplete or unbalanced assessment, depending on the input. Using a meta-modeling approach can address this challenge by quantifying the trustworthiness of software as a "product" and specifying a domain model that captures and conceptualizes trustworthiness. A domain model is a conceptualization of a problem domain in terms of its entities, properties, relationships, and constraints. In software, several domain models exist that are capable of representing and assessing predefined sets of trustworthiness, e.g., PAS 754:2014, QualiPSo [171], and Boland et. al.[172]. All these domain models share a common, while informal (non-machine-readable), structural representation of the trustworthiness they are assessing. This lack of formalism and semantics limits the possible reuse and instantiation for specific trustworthiness assessment contexts.

7.6 Chapter Summary

In summary, this chapter introduced OntTAM, a trustworthiness assessment model which is an instantiation of the SE-EQUAM assessment model. OntTAM takes advantage of the seamless integration of the SBSON, SEON, SEVONT, and MARKOS to provide an automated analysis and assessment of trustworthiness quality attributes. We further presented a concrete instantiation of our assessment model that not only provides a formal modeling of trustworthy quality attributes but can also be extended/customized to specific stakeholder needs. We also illustrated how a concrete instantiation of OntTAM for a small subset of sub-factors, attributes, and measures related to the trustworthiness of reusable components can be created. The measures which we included in the study are: API breaking changes, security vulnerabilities, and license violations.

In the next chapter, we conclude the thesis and discuss some possible future works.

Chapter 8

8 Conclusions and Future Work

In this dissertation, we hypothesized that leveraging build and dependency information in software tasks needs a technology-independent representation of build and dependency management system semantics, integrated with knowledge from other software artifacts.

To validate our thesis, we developed a unified knowledge model for software build and dependency management systems (SBSON). We showed how the integration of additional knowledge sources with SBSON can be performed and illustrate the applicability of our approach in analyzing the impact of code reuse from a dependency management perspective.

8.1 Contributions

In this section we briefly summarize the main contributions of this dissertation compared to the current state of the art.

Modelling Build and Dependency Semantics (Chapter 4). One of the challenges in software traceability is that knowledge about software artifacts is stored in specialized repositories (e.g., build management, versioning, issue trackers), which often have remained information silos – disconnected from each other. Information on how projects are built are stored in similar information silos (e.g., Maven Central, Ruby Gems, and NPM).

In this research, the focus is on the dependency information specified in build systems due to their relevance to support code reuse and global code sharing. We present a formal unified ontological model (SBSON, Software Build System ONtology) which captures concepts and properties for software build systems (Chapter 4). This formal knowledge representation allows us to take advantage of inference services provided by the Semantic Web, providing additional flexibility and benefits such as: a standardized build knowledge representation; cross-artifact analysis, which allows taking advantage of the information in build repositories; and the reuse and sharing of analysis result across artifact and project boundaries.

A Novel Approach to Analyze the Impact of API Breaking Changes (Chapter 5). As discussed throughout the thesis, APIs are commonly used by software developers to reduce development complexity by reusing code developed by third parties or published by the open source community on the Internet. These APIs, however, undergo changes that may break already established contracts, leading to errors and requiring rework in client applications. Identifying the impact of these changes is difficult especially when dealing with transitive API usage across software projects.

We conducted a user survey involving 53 open source developers to gain insights on how they manage API breaking changes. Based on the survey results, we presented a formal unified ontological model which integrates our SBSON model with knowledge about source code usage and changes within the Maven ecosystem. We use this model to identify the potential impact of breaking changes across project boundaries to support library consumers and producers in managing API breaking changes, by taking advantage of SW reasoning services. We present a case study to demonstrate the applicability and flexibility of our approach in supporting library consumers while managing the impacts of breaking changes.

Impact Analysis of Security Vulnerabilities (Chapter 6). Software reuse has increased the threats of sharing software vulnerabilities across project boundaries. Developers are unaware of such security vulnerabilities in their projects, often until a vulnerability is either exploited by attackers or made publicly available by independent security advisory databases. We introduce an integrated dependency and vulnerability knowledge model, SV-AF, in Chapter 6. SV-AF integrates different ontologies such as builds systems ontologies, source code ontologies, version systems ontologies, and vulnerabilities ontologies.

We showed that 750 Maven projects (0.062% of all Maven projects) contain known security vulnerabilities that have been reported in the NVD database [125]. Of these 750 projects, 48.8% suffer even from multiple security vulnerabilities. Our analysis also showed that the same vulnerability can affect multiple releases of a product. The approach presented in this thesis can also be used to identify if the vulnerable source code of a library is indeed being used by a client

[123]. Furthermore, we introduce a vulnerability measure (WVD) that can be used to compare two releases of the same project in terms of their vulnerability impact [147]. The thesis also highlights that this information can be used to guide system update decisions and help avoid the reuse of APIs/components that have known vulnerabilities or are prone to vulnerabilities.

A Model for assessing the Trustworthiness of OSS libraries (Chapter 7). We introduce a novel Ontological Trustworthiness Assessment Model (OntTAM), an extension of the previous generic SE-EQUAM software assessment model [9] (Chapter 7). OntTAM is an integration of our build, source code, vulnerability and license ontologies which supports the automated analysis and assessment of quality attributes related to the trustworthiness of libraries and APIs in open source systems. The main contributions of this assessment model are:

- We introduce new trustworthiness measures, which measure API breaking changes, security vulnerabilities, and license violations.
- We perform several case studies to illustrate how our approach provides developers with additional insights on the potential impact of reused libraries and APIs on the quality and trustworthiness of their project.

Impact Analysis of License Violations (Section 7.4.3). The reuse of libraries leads to hierarchies of libraries and license dependencies. These libraries' licenses must be compatible and compliant with each other. License violations and incompatibilities are an often-overlooked factor when recommending APIs and therefore can significantly impact the trustworthiness of software systems. We extend the MARKOS license ontology [10] with semantic rules for three categories of license violations, and perform a study on the Maven ecosystem to identify direct and transitive license violations [147]. The study identified over 131,000 simple violations and 943,000 transitive license violations. Such findings suggest the need for additional automated support for recommending trustworthy libraries.

8.2 Future Work

8.2.1 Current Limitations

Quality of our Ontology Design. One of the major benefits of our approach is its ability to seamlessly integrate and reuse ontologies. However, assessing the quality of our ontology designs is an inherently difficult problem since what constitutes quality depends on different non-functional requirements (e.g., reuse, usability, extensibility, expressiveness and reasoning support). We partly address this threat by using existing reasoners (such as Pellet, Hermit, and JFact) and tools (OOPS![58] and the Neon Toolkit[59]) to check our ontology design for taxonomic, syntactical and consistency problems. To determine if our ontology constraints were sufficient to identify incorrect data, we incrementally populated the ontologies with facts during the evaluation process. While the reasoners did not report any inconsistencies in our ontologies, OOPS! reported a few problems in our ontologies which violated some of the design rules in OOPS! rule catalog. The identified violations were a result of missing license information and annotations (such as <rdfs:label> and <rdfs:comment>) for some of our ontology elements.

Another potential threat to our approach is whether the set of concepts we considered is enough to capture the semantics of the analyzed domains. There is always a trade-off in the design of knowledge bases in terms of their expressivity and their usefulness; an equilibrium should be established between the amount of information that is sufficient to accomplish a task and the granularity of the knowledge that should be available to produce useful results. We addressed this threat by showing that our modeled concepts are enough to provide flexible analysis services through the described case study experiments.

Generalizability. The case studies described in this thesis are limited in their scope to open source Java projects in the Maven repository, and the results obtained might not be applicable to other programming languages or build repositories. Given that our modeling approach is based on different levels of abstraction, we also abstract common aspects of source code and build dependencies in our knowledge model. We do model the domain of object-oriented

[58] http://oops.linkeddata.es/advanced.jsp
[59] http://neon-toolkit.org/wiki/Download/2.5.2.html

programming languages, software vulnerabilities, software licenses, and build repositories as individual domains of discourse in the domain-specific layer of our knowledge model.

8.2.2 Opportunities for Future Research

The presented research involves different areas of computer science, including SW technologies, knowledge modeling, mining software repositories, and source code analysis. This diversity of topics also leads to multiple research directions in which the work presented in this thesis can be extended as part of future work.

Integrating Crowd Based Knowledge Sources. Changes to the software development process (such as increased collaboration and agile work habits) have made the Internet a great source of information, documentation and explanations to support the work context of developers [189]. These crowd-based information sources (e.g., blogs, online video tutorials, Q/A forums) contain important information which are often fragmented. One interesting avenue for future research is the mining, modeling, and integration of crowd cased knowledge related to code reuse.

As part of our ongoing research we have already proposed an approach which integrates online screencasts with known security issues. More specifically, we leverage audio, video (textual cues in image frames) and metadata from screencasts published on YouTube and integrate this knowledge with software dependency and security related knowledge from our existing SV-AF approach. We establish bi-directional traceability links from screencasts to NVD security vulnerabilities and infer indirect traceability links between screencasts and Maven project dependencies, which takes advantage of our existing traceability links (in SV-AF) between NVD and Maven Central. We argue that these links can be used to provide practitioners with additional insights in comprehending the potential impact of using vulnerable projects in their projects or how screencasts address these known security issues. Traceability links between screencasts and vulnerability reports are inferred by (1) identifying vulnerability references such as the CVE ID and CWE ID in the title, description, speech, or image frames of the screencasts, and (2) using the BM25 probabilistic relevance model [190], a popular model used in Information Retrieval (IR), to rank a set of vulnerability reports based on their relevance to words in a given screencast. Our initial experiments on 48 selected vulnerability related

YouTube videos showed that our approach can successfully link relevant vulnerabilities and screencasts with an average precision of 98% and an average recall of 54% when vulnerability identifiers (CVE ID) are explicitly mentioned in the videos. When no direct reference to a CVE ID exists in the screencast, our approach was still able to link video-vulnerability descriptions, with up to 100% of the time relevant links being ranked in the 2nd position of our results set.

Having this knowledge integration not only provides developers with direct access to vulnerability information described in a screencast content, but also allows us to link vulnerability descriptions to relevant screencasts and dependency information. In addition, our approach also allows developers to identify screencasts that demonstrate such attacks and provides developers who are indirectly using vulnerable libraries in their project (e.g., through Maven dependencies) with insights on how to reduce the potential impact of being directly or indirectly exposed to a vulnerability.

As part of our future work, we plan to extend our modeling approach to integrate videos and their content with other software artifacts and to conduct larger case studies to further improve the generalizability of our approach. We also plan to include knowledge from other crowd-based information sources such as blogs and Q/A forums (e.g., StackOverflow).

Build Quality and the Performance of Continuous Integration. Continuous integration (CI) platforms automate the process of building and testing these projects. Despite CI's many benefits and wide popularity, CI's process can take a very long time to complete and can be particularly problematic when builds fail. Research has tried to understand why builds fail [191], and even try to predict the build results [192]. However, very few studies tried to improve the efficiency of the CI process. We will work on extending our assessment framework [147] to evaluates the quality of a project's build at commit time. Furthermore, to make the process fast and efficient, it is important for the approach to perform this analysis incrementally on only the new changes to the project. Other interesting aspects of build quality considered for future research include build clones and unused build configurations.

Optimizing our Knowledge Model. An impending threat to knowledge-based systems using graph structures for information modeling are inefficient, slow query times compared to relational databases. If our knowledge base is expected to integrate the knowledge of existing global (dependency related) software artifacts, a detailed study of different optimization techniques over currently used graph-based query languages, such as SPARQL, is crucial.

Bibliography

[1] J. Z. Gao, C. Chen, Y. Toyoshima, and D. K. Leung, "Engineering on the Internet for global software production," *Computer (Long. Beach. Calif).*, vol. 32, no. 5, pp. 38–47, May 1999.

[2] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," *Proc. Jt. Int. Annu. ERCIM Work. Princ. Softw. Evol. Softw. Evol.*, pp. 57–62, 2009.

[3] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Softw.*, vol. 26, no. 6, pp. 27–34, Nov. 2009.

[4] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *J. Syst. Softw.*, vol. 145, pp. 164–179, Nov. 2018.

[5] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel, "Tracing software build processes to uncover license compliance inconsistencies," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, 2014, pp. 731–742.

[6] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining Co-change Information to Understand When Build Changes Are Necessary," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 241–250.

[7] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 311–320.

[8] S. McIntosh *et al.*, "Collecting and leveraging a benchmark of build system clones to aid in quality assessments," *Companion Proc. 36th Int. Conf. Softw. Eng. - ICSE Companion 2014*, pp. 145–154, 2014.

[9] A. Hmood, I. Keivanloo, and J. Rilling, "SE-EQUAM - An evolvable quality metamodel," *Proc. - Int. Comput. Softw. Appl. Conf.*, pp. 334–339, Jul. 2012.

[10] G. Bavota *et al.*, "The market for open source: An intelligent virtual open source marketplace," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 399–402.

[11] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, 2017, pp. 385–395.

[12] F. L. de la Mora and S. Nadi, "An Empirical Study of Metric-based Comparisons of Software Libraries," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE'18*, 2018, pp. 22–31.

[13] F. Thung, “API recommendation system for software development,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016, pp. 896–899.

[14] M. M. Rahman, C. K. Roy, and D. Lo, “RACK: Automatic API Recommendation Using Crowdsourced Knowledge,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 349–359.

[15] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?,” *Empir. Softw. Eng.*, vol. 23, no. 1, pp. 384–417, Feb. 2018.

[16] C. Teyton, J. R. Falleri, and X. Blanc, “Mining library migration graphs,” *Proc. - Work. Conf. Reverse Eng. WCRE*, pp. 289–298, 2012.

[17] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, “How the Apache community upgrades dependencies: an evolutionary study,” *Empir. Softw. Eng.*, vol. 20, no. 5, pp. 1275–1317, 2014.

[18] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, “How to break an API: cost negotiation and community values in three software ecosystems,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, 2016, pp. 109–120.

[19] A. Decan, T. Mens, and M. Claes, “An empirical comparison of dependency issues in OSS packaging ecosystems,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 2–12.

[20] A. Decan, T. Mens, and E. Constantinou, “On the impact of security vulnerabilities in the npm package dependency network,” in *Proceedings of the 15th International Conference on Mining Software Repositories - MSR ’18*, 2018, pp. 181–191.

[21] M. Cadariu, E. Bouwers, J. Visser, and A. Van Deursen, “Tracking known security vulnerabilities in proprietary software systems,” *2015 IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2015 - Proc.*, pp. 516–519, 2015.

[22] D. M. German, M. Di Penta, and J. Davies, “Understanding and Auditing the Licensing of Open Source Software Distributions,” in *2010 IEEE 18th International Conference on Program Comprehension*, 2010, pp. 84–93.

[23] R. G. Kula, D. M. German, T. Ishio, A. Ouni, and K. Inoue, “An exploratory study on library aging by monitoring client usage in a software ecosystem,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 407–411.

[24] B. Motik, I. Horrocks, and U. Sattler, “Bridging the gap between OWL and relational databases,” *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 7, no. 2, pp. 74–89, Apr. 2009.

[25] M. Würsch, G. Reif, S. Demeyer, and H. C. Gall, “Fostering Synergies – How Semantic Web Technology could influence Software Repositories,” *Scenario*, pp. 45–48, 2010.

[26] J. RILLING, R. WITTE, P. SCHUEGERL, and P. CHARLAND, “BEYOND INFORMATION SILOS — AN OMNIPRESENT APPROACH TO SOFTWARE EVOLUTION,” *Int. J. Semant. Comput.*, vol. 02, no. 04, pp. 431–468, Dec. 2008.

[27] T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” *Sci. Am.*, vol. 284, no. 5, pp. 34–

43, May 2001.

[28] B. Berendt, A. Hotho, D. Mladenic, M. Someren, M. Spiliopoulou, and G. Stumme, “Web Mining: From Web to Semantic Web: First European Web Mining Forum, EWMF 2003, Cavtat-Dubrovnik, Croatia, September 22, 2003, Invited and Selected Revised Papers,” B. Berendt, A. Hotho, D. Mladenič, M. Someren, M. Spiliopoulou, and G. Stumme, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–22.

[29] T. Segaran, C. Evans, and J. Taylor, *Programming the semantic web*. “ O’Reilly Media, Inc.,” 2009.

[30] T. Heath and C. Bizer, “Linked Data: Evolving the Web into a Global Data Space,” *Synth. Lect. Semant. Web Theory Technol.*, vol. 1, no. 1, pp. 1–136, Feb. 2011.

[31] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*, vol. 32, no. 9/10. 2010.

[32] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, “The evolution of project inter-dependencies in a software ecosystem: The case of apache,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 280–289, 2013.

[33] D. Binkley, “Source Code Analysis: A Road Map,” in *Future of Software Engineering (FOSE ’07)*, 2007, pp. 104–119.

[34] M. Würsch, G. Ghezzi, M. Hert, G. Reif, and H. C. Gall, “SEON: a pyramid of ontologies for software evolution and its applications,” *Computing*, vol. 94, no. 11, pp. 857–885, Nov. 2012.

[35] B. Decker, E. Ras, J. Rech, B. Klein, and C. Hoecht, “Self-organized reuse of software engineering knowledge supported by semantic wikis,” in *Proceedings of the Workshop on Semantic Web Enabled Software Engineering (SWESE)*, 2005, p. 76.

[36] Y. Zhang, J. Rilling, and V. Haarslev, “An Ontology-Based Approach to Software Comprehension - Reasoning about Security Concerns,” in *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*, 2006, pp. 333–342.

[37] B. Wouters, D. Deridder, and E. Van Paesschen, “The use of ontologies as a backbone for use case management,” in *European Conference on Object-Oriented Programming (ECOOP 2000), Workshop: Objects and Classifications, a natural convergence*, 2000, vol. 182.

[38] U. Nonnenmann and J. K. Eddy, “KITSS-a functional software testing system using a hybrid domain model,” in *Proceedings Eighth Conference on Artificial Intelligence for Applications*, 2003, pp. 136–142.

[39] A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, and C. Welty, “Supporting online problem-solving communities with the semantic web,” in *Proceedings of the 15th international conference on World Wide Web - WWW ’06*, 2006, pp. 575–584.

[40] H.-J. Happel, A. Korthaus, S. Seedorf, and P. Tomczyk, “KOntoR: An Ontology-enabled Approach to Software Reuse,” in *In: Proc. Of The 18Th Int. Conf. On Software Engineering And Knowledge Engineering*, 2006.

[41] D. Jin and J. R. Cordy, “A Service Sharing Approach to Integrating Program Comprehension Tools,” in *Proc. European Software Engineering Conference (ESEC) / ACM Symposium on the Foundations of Software Engineering (FSE) 2003 Workshop on Tool Integration in System*

Development, 2003, pp. 73–78.

[42] D. Hyland-Wood, D. Carrington, and S. Kaplan, "Toward a Software Maintenance Methodology using Semantic Web Techniques," in *2006 Second International IEEE Workshop on Software Evolvability (SE'06)*, 2006, pp. 23–30.

[43] M. F. Bertoa, A. Vallecillo, and F. García, "An Ontology for Software Measurement," in *Ontologies for Software Engineering and Software Technology*, Springer Berlin Heidelberg, 2006, pp. 175–196.

[44] R. Witte, Y. Zhang, and J. Rilling, "Empowering software maintainers with semantic web technologies," *Eur. Conf. Semant. Web Res. Appl.*, pp. 37–52, 2007.

[45] L. Yu, J. Zhou, Y. Yi, P. Li, and Q. Wang, "Ontology Model-Based Static Analysis on Java Programs," in *2008 32nd Annual IEEE International Computer Software and Applications Conference*, 2008, pp. 92–99.

[46] C. Kiefer, A. Bernstein, and J. Tappolet, "Mining Software Repositories with iSPAROL and a Software Evolution Ontology," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 2007, pp. 10–10.

[47] A. Iqbal, G. Tummarello, M. Hausenblas, and O.-E. Ureche, "LD2SD: linked data driven software development," in *International Conference on Software Engineering & Knowledge Engineering*, 2009.

[48] A. E. Hassan and R. C. Holt, "The top ten list: dynamic fault prediction," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 263–272.

[49] M. W. Godfrey *et al.*, "Future of Mining Software Archives: A Roundtable," *IEEE Softw.*, vol. 26, no. 1, pp. 67–70, Jan. 2009.

[50] A. Hassan, "Mining Software Repositories to Assist Developers and Support Managers," in *2006 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 339–342.

[51] A. E. Hassan, "The road ahead for Mining Software Repositories," in *2008 Frontiers of Software Maintenance*, 2008, pp. 48–57.

[52] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.

[53] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 2008, p. 181.

[54] P. C. Rigby and A. E. Hassan, "What can OSS mailing lists tell us? A preliminary psychometric text analysis of the Apache developer mailing list," *Proc. - ICSE 2007 Work. Fourth Int. Work. Min. Softw. Repos. MSR 2007*, 2007.

[55] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open borders? Immigration in open source projects," *Proc. - ICSE 2007 Work. Fourth Int. Work. Min. Softw. Repos. MSR 2007*, 2007.

[56] a. Mockus, P. Z. P. Zhang, and P. L. Li, "Predictors of customer perceived software quality," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005,

pp. 225–233.

[57] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05*, 2005, p. 48.

[58] J. I. Maletic and M. L. Collard, "Supporting source code difference analysis," in *IEEE International Conference on Software Maintenance, ICSM*, 2004, pp. 210–219.

[59] R. W. Selby, "Enabling reuse-based software development of large-scale systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 495–510, Jun. 2005.

[60] M. Ohira, "Empirical project monitor: a tool for mining multiple project data," in *"International Workshop on Mining Software Repositories (MSR 2004)" W17S Workshop - 26th International Conference on Software Engineering*, 2004, vol. 2004, pp. 42–46.

[61] M. Ohira, N. Ohsugi, T. Ohoka, and K. Matsumoto, "Accelerating cross-project knowledge collaboration using collaborative filtering and social networks," in *Proceedings of the 2005 international workshop on Mining software repositories - MSR '05*, 2005, pp. 1–5.

[62] R. J. Sandusky, "Bug report networks: varieties, strategies, and impacts in a F/OSS development community," in *"International Workshop on Mining Software Repositories (MSR 2004)" W17S Workshop - 26th International Conference on Software Engineering*, 2004, vol. 2004, pp. 80–84.

[63] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," *Proceeding 28th Int. Conf. Softw. Eng. - ICSE '06*, vol. 2006, p. 361, 2006.

[64] H. Kagdi, S. Yusuf, and J. I. Maletic, "Mining sequences of changed-files from version histories," in *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, 2006, p. 47.

[65] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, Sep. 2004.

[66] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005.

[67] B. Livshits and T. Zimmermann, "DynaMine," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, 2005, p. 296.

[68] C. C. Williams and J. K. Hollingsworth, "Recovering system specific rules from software repositories," in *Proceedings of the 2005 international workshop on Mining software repositories - MSR '05*, 2005, pp. 1–5.

[69] F. Van Rysselberghe, "Mining version control systems for FACs (frequently applied changes)," in *"International Workshop on Mining Software Repositories (MSR 2004)" W17S Workshop - 26th International Conference on Software Engineering*, 2004, vol. 2004, pp. 48–52.

[70] D. M. German, "An empirical study of fine-grained software modifications," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 316–325.

[71] C. Görg and P. Weißgerber, "Error detection by refactoring reconstruction," in *Proceedings of the 2005 international workshop on Mining software repositories - MSR '05*, 2005, pp. 1–5.

[72] A. Chen *et al.*, “CVSSearch: Searching through source code using CVS comments,” in *IEEE International Conference on Software Maintenance, ICSM*, 2001, pp. 364–375.

[73] M. Kim and D. Notkin, “Using a clone genealogy extractor for understanding and supporting evolution of code clones,” in *Proceedings of the 2005 international workshop on Mining software repositories - MSR ’05*, 2005, pp. 1–5.

[74] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider, “Answering questions about unanswered questions of stack overflow,” *Proc. 10th Work. Conf. Min. Softw. Repos.*, pp. 97–100, 2013.

[75] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, “What makes a good code example?: A study of programming Q&amp;A in StackOverflow,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 25–34.

[76] B. Bazelli, A. Hindle, and E. Stroulia, “On the Personality Traits of StackOverflow Users,” in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 460–463.

[77] S. Wang, D. Lo, and L. Jiang, “An empirical study on developer interactions in StackOverflow,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC ’13*, 2013, p. 1019.

[78] A. Barua, S. W. Thomas, and A. E. Hassan, “What are developers talking about? An analysis of topics and trends in Stack Overflow,” *Empir. Softw. Eng.*, vol. 19, no. 3, pp. 619–654, Jun. 2014.

[79] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, “A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance,” *Empir. Softw. Eng.*, vol. 20, no. 6, pp. 1587–1633, 2014.

[80] P. Smith, *Software Build Systems – Principles and Experience*, 1st ed. Addison Wesley, 2011.

[81] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, “The Evolution of the Linux Build System,” *Electron. Commun. EASST*, vol. 8, 2007.

[82] B. Adams, “Co-evolution of source code and the build system,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 461–464, 2009.

[83] Godfrey and Qiang Tu, “Evolution in open source software: a case study,” in *Proceedings International Conference on Software Maintenance ICSM-94*, 2000, pp. 131–142.

[84] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, “An empirical study of build maintenance effort,” *2011 33rd Int. Conf. Softw. Eng.*, pp. 141–150, 2011.

[85] S. Raemaekers, A. Van Deursen, and J. Visser, “Semantic versioning versus breaking changes: A study of the maven repository,” *Proc. - 2014 14th IEEE Int. Work. Conf. Source Code Anal. Manip. SCAM 2014*, pp. 215–224, 2014.

[86] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits, “Automated migration of build scripts using dynamic analysis and search-based refactoring,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA ’14*, 2014, pp. 599–616.

[87] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, “Fault localization for build code errors in makefiles,” in *Companion Proceedings of the 36th International Conference on Software*

Engineering - ICSE Companion 2014, 2014, pp. 600–601.

[88] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, “Build code analysis with symbolic evaluation,” *Proc. - Int. Conf. Softw. Eng.*, pp. 650–660, 2012.

[89] C. Dietrich, R. Tartler, and W. S. D. Lohmann, “A Robust Approach for Variability Extraction from the Linux Build System,” pp. 21–30, 2012.

[90] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski, “Feature-to-Code Mapping in Two Large Product Lines,” 2010, pp. 498–499.

[91] S. Nadi and R. Holt, “The Linux kernel : a case study of build system variability,” 2013.

[92] S. Zhou, J. Al-Kofahi, T. N. Nguyen, C. Kastner, and S. Nadi, “Extracting Configuration Knowledge from Build Files with Symbolic Analysis,” *2015 IEEE/ACM 3rd Int. Work. Release Eng.*, pp. 20–23, 2015.

[93] B. Motik, A. Maedche, and R. Volz, “A Conceptual Modeling Approach for Semantics-Driven Enterprise Applications,” *Move to Meaningful Internet Syst. 2002 CoopIS, DOA, ODBASE*, vol. 2519, pp. 1082–1099, 2000.

[94] T. R. Gruber, “Toward principles for the design of ontologies used for knowledge sharing,” *Int. J. Hum. Comput. Stud.*, vol. 43, no. 5–6, pp. 907–928, 1995.

[95] N. Noy and D. McGuinness, “Ontology Development 101: A Guide to Creating Your First Ontology,” 2001.

[96] P. E. van der Vet and N. J. I. Mars, “Bottom-up construction of ontologies,” *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 4, pp. 513–526, 1998.

[97] M. Uschold and M. Gruninger, “Ontologies: principles, methods and applications,” *Knowl. Eng. Rev.*, vol. 11, no. 02, p. 93, Jun. 1996.

[98] S. S. Alqahtani, E. E. Eghan, and J. Rilling, “SE-GPS,” 2015. [Online]. Available: http://aseg.encs.concordia.ca/segps/. [Accessed: 05-Jan-2019].

[99] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, “AURA: a hybrid approach to identify framework evolution,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE ’10*, 2010, vol. 1, p. 325.

[100] G. Brito, A. Hora, M. T. Valente, and R. Robbes, “Do Developers Deprecate APIs with Replacement Messages? A Large-Scale Analysis on Java Systems,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 360–369.

[101] A. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse, “How do developers react to API evolution? A large-scale empirical study,” *Softw. Qual. J.*, Oct. 2016.

[102] R. Robbes, M. Lungu, and D. Röthlisberger, “How do developers react to API deprecation?,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE ’12*, 2012, p. 1.

[103] M. Kim, D. Notkin, and D. Grossman, “Automatic Inference of Structural Changes for Matching across Program Versions,” in *29th International Conference on Software Engineering (ICSE’07)*,

2007, pp. 333–343.

[104] P. Weissgerber and S. Diehl, "Identifying Refactorings from Source-Code Changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 231–240.

[105] B. Dagenais and M. P. Robillard, "Recommending Adaptive Changes for Framework Evolution," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 1–35, Sep. 2011.

[106] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 2008, p. 471.

[107] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 138–147.

[108] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empir. Softw. Eng.*, vol. 21, no. 6, pp. 2366–2412, Dec. 2016.

[109] J. Singer, S. E. Sim, and T. C. Lethbridge, "Software Engineering Data Collection for Field Studies," in *Guide to Advanced Empirical Software Engineering*, London: Springer London, 2008, pp. 9–34.

[110] D. Movshovitz-Attias, S. E. Whang, N. Noy, and A. Halevy, "Discovering Subsumption Relationships for Web-Based Ontologies," in *Proceedings of the 18th International Workshop on Web and Databases - WebDB'15*, 2010, pp. 62–69.

[111] Y. Wang *et al.*, "Do the dependency conflicts in my project matter?," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, 2018, pp. 319–330.

[112] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*, 2011, pp. 1317–1324.

[113] J. Businge, A. Serebrenik, and M. G. J. van den Brand, "Eclipse API usage: the good and the bad," *Softw. Qual. J.*, vol. 23, no. 1, pp. 107–141, Mar. 2015.

[114] B. E. Cossette and R. J. Walker, "Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries," *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, pp. 55:1--55:11, 2012.

[115] P. Kapur, B. Cossette, and R. J. Walker, "Refactoring references for library migration," *ACM SIGPLAN Not.*, vol. 45, no. 10, p. 726, 2010.

[116] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *J. Softw. Maint. Evol. Res. Pract.*, vol. 18, no. 2, pp. 83–107, Mar. 2006.

[117] S. Raemaekers, A. Van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 378–387, 2012.

[118] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub Meets CRAN: An Analysis of

Inter-Repository Package Dependency Problems," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 493–504.

[119] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, S. Zacchiroli, and A. P. S. Distributions, "Why Do Software Packages Conflict ?," pp. 141–150, 2012.

[120] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," *Asp. Secur. Inc*, pp. 1–26, 2012.

[121] OWASP, "Top 10-2017 A9-Using Components with Known Vulnerabilities," 2018. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities. [Accessed: 05-Jul-2019].

[122] B. Liu, L. Shi, Z. Cai, and M. Li, "Software Vulnerability Discovery Techniques: A Survey," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, 2012, pp. 152–156.

[123] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach," in *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*, 2017, pp. 80–91.

[124] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "SV-AF - A Security Vulnerability Analysis Framework," in *IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 219–229.

[125] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Tracing known security vulnerabilities in software repositories - A Semantic Web enabled modeling approach," *Sci. Comput. Program.*, vol. 121, pp. 153–175, Feb. 2016.

[126] N. McNeil, R. A. Bridges, M. D. Iannacone, B. Czejdo, N. Perez, and J. R. Goodall, "PACE: Pattern Accurate Computationally Efficient Bootstrapping for Timely Discovery of Cyber-security Concepts," in *2013 12th International Conference on Machine Learning and Applications*, 2013, pp. 60–65.

[127] S. S. Alqahtani, "Enhancing Trust–A Unified Meta-Model for Software Security Vulnerability Analysis," Concordia University, 2018.

[128] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios, "k-nearest neighbors in uncertain graphs," *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 997–1008, Sep. 2010.

[129] A. Kimmig, S. Bach, M. Broecheler, B. Huang, and L. Getoor, "A short introduction to probabilistic soft logic," in *Proceedings of the NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012, pp. 1–4.

[130] NIST, "National Vulnerability Database," 2007. .

[131] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Empir. Softw. Eng.*, Dec. 2015.

[132] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," *2015 IEEE 31st Int. Conf. Softw. Maint. Evol. ICSME 2015 - Proc.*, pp. 411–420, 2015.

[133] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining API popularity," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6303 LNCS, pp. 173–180, 2010.

[134] M. Hirzel, D. Von Dincklage, A. Diwan, and M. Hind, "Fast online pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 2, pp. 11–66, Apr. 2007.

[135] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, 1999, pp. 50–59.

[136] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*, 1993, pp. 232–245.

[137] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "The bug catalog of the maven ecosystem," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 372–375.

[138] C. V Saini, Vaibhav and Sajnani, Hitesh and Ossher, Joel and Lopes, "A dataset for maven artifacts and bug patterns found in them," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 416--419.

[139] A. Pletea, Daniel and Vasilescu, Bogdan and Serebrenik, "Security and emotion: sentiment analysis of security discussions on GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 348–351.

[140] T. Gegick, Michael and Rotella, Pete and Xie, "Identifying security bug reports via text mining: An industrial case study," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 2010, pp. 11--20.

[141] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*, 2007, pp. 529–540.

[142] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," *Proc. - Work. Conf. Reverse Eng. WCRE*, no. October, pp. 182–191, 2013.

[143] A. Hora, A. Hora, and M. T. Valente, "apiwave : Keeping Track of API Popularity and Migration," no. JANUARY, pp. 321–323, 2015.

[144] F. McCarey, M. Ó. Cinnéide, and N. Kushmerick, "Rascal: A Recommender Agent for Agile Reuse," *Artif. Intell. Rev.*, vol. 24, no. 3–4, pp. 253–276, Nov. 2005.

[145] D. L. Parnas, "Software aging," in *ICSE '94 Proceedings of the 16th international conference on Software engineering*, 1994, pp. 279–287.

[146] F. S. Foundation, "Various Licenses and Comments About Them," *GNU Project*, 2014. [Online]. Available: https://www.gnu.org/licenses/license-list.en.html.

[147] E. E. Eghan, S. S. Alqahtani, C. Forbes, and J. Rilling, "API trustworthiness: an ontological approach for software library adoption," *Softw. Qual. J.*, pp. 1–46, 2019.

[148] O. S. Initiative, "The Open Source Definition," *Open Source Software*, 2007. [Online]. Available:

https://opensource.org/osd. [Accessed: 06-Jul-2019].

[149] G. M. Kapitsaki, F. Kramer, and N. D. Tselikas, “Automating the license compatibility process in open source software with SPDX,” *J. Syst. Softw.*, vol. 131, pp. 386–401, Sep. 2017.

[150] I. GitHub, “Choose a License,” 2008. [Online]. Available: http://creativecommons.org/license/. [Accessed: 06-Jul-2019].

[151] O. Seneviratne, L. Kagal, D. Weitzner, H. Abelson, T. Berners-Lee, and N. Shadbolt, “Detecting creative commons license violations on images on the world wide web,” *WWW2009, April*, 2009.

[152] L. An, O. Mlouki, F. Khomh, and G. Antoniol, “Stack Overflow: A code laundering platform?,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 283–293.

[153] Software Freedom Law Center, “Best Buy, Samsung, Westinghouse, And Eleven Other Brands Named In SFLC Lawsuit,” 2009. [Online]. Available: http://www.softwarefreedom.org/news/2009/dec/14/busybox-gpl-lawsuit/. [Accessed: 06-Jul-2019].

[154] Software Freedom Law Center, “Motion Against Westinghouse Digital Electronics in GPL Compliance Lawsuit,” 2010. [Online]. Available: https://www.softwarefreedom.org/news/2010/jun/07/motion-against-westinghouse-digital-electronics-gp/. [Accessed: 06-Jul-2019].

[155] B. A. Kitchenham and J. G. Walker, “A quantitative approach to monitoring software development,” *Softw. Eng. J.*, vol. 4, no. 1, p. 2, 2010.

[156] B. Kitchenham and S. L. Pfleeger, “Software quality: the elusive target,” *IEEE Softw.*, vol. 13, no. 1, pp. 12–21, 1996.

[157] D. Hoyle, “Chapter 2 - Defining and Characterizing Quality,” in *ISO 9000 Quality Systems Handbook - updated for the ISO 9001:2008 standard (Sixth Edition)*, Sixth Edit., D. Hoyle, Ed. Oxford: Butterworth-Heinemann, 2009, pp. 23–37.

[158] J. A. McCall, P. K. Richards, and G. F. Walters, “Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality,” 1977.

[159] A. Bergel *et al.*, “SQUALE - Software QUALity Enhancement,” in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 285–288.

[160] A. Hmood, Philipp Schugerl, J. Rilling, and Philippe Charland, “OntEQAM – A Methodology for Assessing Evolvability as a Quality Factor in Software Ecosystems,” in *Defence R&D Canada - Valcartier, Valcartier QUE (CAN)*, 2010, p. 8.

[161] S. Seedorf and F. F. I. U. Mannheim, “Applications of Ontologies in Software Engineering,” in *In 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*, 2006.

[162] H. Kagdi, M. L. Collard, and J. I. Maletic, “Comparing Approaches to Mining Source Code for Call-Usage Patterns,” in *Fourth International Workshop on Mining Software Repositories (MSR’07:ICSE Workshops 2007)*, 2007, pp. 20–26.

[163] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone

detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[164] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev, "Ontological approach for the semantic recovery of traceability links between software artefacts," *IET Softw.*, vol. 2, no. 3, p. 185, 2008.

[165] I. Keivanloo, C. Forbes, J. Rilling, and P. Charland, "Towards sharing source code facts using linked data," *Proceeding 3rd Int. Work. Search-driven Dev. users, infrastructure, tools, Eval. - SUITE '11*, pp. 25–28, 2011.

[166] L. A. Zadeh, "The concept of a linguistic variable and its application to approximate reasoning-III," *Inf. Sci. (Ny).*, vol. 9, no. 1, pp. 43–80, Jan. 1975.

[167] I. E. Commission, "Programmable Controllers - Part 7: Fuzzy Control Programming," 2000.

[168] P. Cingolani and J. Alcala-Fdez, "jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation," in *2012 IEEE International Conference on Fuzzy Systems*, 2012, pp. 1–8.

[169] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, "The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation," in *Open Source Development, Communities and Quality*, Boston, MA: Springer US, 2008, pp. 237–248.

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

[171] V. del Bianco, L. Lavazza, S. Morasca, and D. Taibi, "Quality of Open Source Software: The QualiPSo Trustworthiness Model," 2009, pp. 199–212.

[172] T. Boland, C. Cleraux, and E. Fong, "Toward a Preliminary Framework for Assessing the Trustworthiness of Software," *Natl. Inst. Stand. Technol.*, no. September, pp. 1–31, 2010.

[173] H. Jiang, J. Zhang, Z. Ren, and T. Zhang, "An Unsupervised Approach for Discovering Relevant Tutorial Fragments for APIs," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 38–48.

[174] W. Maalej and M. P. Robillard, "Patterns of Knowledge in API Reference Documentation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1264–1282, Sep. 2013.

[175] M. Di Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol, "An Exploratory Study of the Evolution of Software Licensing," *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. - ICSE '10*, vol. 1, pp. 1–10, 2010.

[176] Y. Wu, Y. Manabe, T. Kanda, D. M. German, and K. Inoue, "A Method to Detect License Inconsistencies in Large-Scale Open Source Projects," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 324–333.

[177] H. Zhong and H. Mei, "An Empirical Study on API Usages," *IEEE Trans. Softw. Eng. (Early Access)*, pp. 1–1, 2017.

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

[179] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, “Finding software license violations through binary code clone detection,” in *Proceeding of the 8th working conference on Mining software repositories - MSR ’11*, 2011, pp. 63–72.

[180] D. M. German and A. E. Hassan, “License integration patterns: Addressing license mismatches in component-based development,” in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 188–198.

[181] D. Taibi, “Defining an Open Source Software Trustworthiness Model,” *Proc. 3rd Int. Dr. Symp. Emperical Softw. Eng.*, p. 4, 2008.

[182] D. Larson and K. Miller, “Silver bullets for little monsters: making software more trustworthy,” *IT Prof.*, vol. 7, no. 2, pp. 9–13, Jan. 2005.

[183] T. Tan, M. He, Y. Yang, Q. Wang, and M. Li, “An Analysis to Understand Software Trustworthiness,” in *2008 The 9th International Conference for Young Computer Scientists*, 2008, pp. 2366–2371.

[184] S. L. Pfleeger, “Measuring software reliability,” *IEEE Spectr.*, vol. 29, no. 8, pp. 56–60, Aug. 1992.

[185] Y. Yang, Q. Wang, and M. Li, “Process Trustworthiness as a Capability Indicator for Measuring and Improving Software Trustworthiness,” 2009, pp. 389–401.

[186] T. Rhodes, F. Boland, E. Fong, and M. Kass, “Software Assurance using Structured Assurance Case Models,” *J. Res. Natl. Inst. Stand. Technol.*, vol. 115, no. 3, 2010.

[187] R. Land, D. Sundmark, F. Lüders, I. Krasteva, and A. Causevic, “Reuse with Software Components - A Survey of Industrial State of Practice,” *Form. Found. Reuse Domain Eng.*, pp. 150–159, 2009.

[188] C. Ayala, X. Franch, R. Conradi, J. Li, and D. Cruzes, “Developing Software with Open Source Software Components,” in *Finding Source Code on the Web for Remix and Reuse*, New York, NY: Springer New York, 2013, pp. 167–186.

[189] P. Moslehi, B. Adams, and J. Rilling, “On mining crowd-based speech documentation,” in *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR ’16*, 2016, pp. 259–268.

[190] C. D. Manning, P. Raghavan, and H. Schütze, *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[191] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, “Programmers’ build errors: a case study (at google),” *Proc. 36th Int. Conf. Softw. Eng. - ICSE 2014*, no. Section 2, pp. 724–734, 2014.

[192] A. E. Hassan and Z. Ken, “Using decision trees to predict the certification result of a build,” *Proc. - 21st IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2006*, pp. 189–198, 2006.

Appendix A: Referenced Ontologies

The following table provides the ontology description and namespaces used in this dissertation, as well as their corresponding URIs.

Ontology	Namespace	URI	Description
GENERAL	main	http://aseg.cs.concordia.ca/segps/ontologies/general/2015/02/main.owl#	Our general layer ontology
MARKOS	markos	http://www.markosproject.eu/ontologies/osslicenses	The MARKet for Open Source license ontology
MEASUREMENT	measure	http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2015/02/measurement.owl#	Our measurement ontology
OLO	olo	http://purl.org/ontology/olo/core#	The OrderedList Ontology
ONTTAM	onttam	http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2017/09/onttam.owl#	Our trustworthiness assessment ontology
OWL	owl	http://www.w3.org/2002/07/owl#	Web Ontology Language
RDF	rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#	Resource Description Framework
SBSON	sbson	http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#	Our Software Build System ONtology
SEON	seon	http://se-on.org/ontologies/general/2012/02/main.owl#	The Software Evolution ONtology
SEON-HISTORY	version	http://se-on.org/ontologies/domain-specific/2012/02/history.owl#	SEON's versioning domain ontology
SEQUAM	sequam	http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2017/09/sequam.owl#	The quality assessment ontology
SEVONT	sevont	http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2015/02/vulnerabilities.owl#	The SEcurity Vulnerability ONTolgy
SOCON	code	http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/code.owl#	Our SOurce Code ONtology

Appendix B: User Survey Questionnaire

Part I: Background

1. How best would you describe yourself?
   a. Undergraduate Student
   b. Graduate Student
   c. Academic Researcher
   d. Industrial Researcher
   e. Industrial Developer
   f. Freelance Developer
   g. Other: ___________________
2. How many years of software development or maintenance experience do you have?
   a. < 1 year
   b. 1 -2 years
   c. 2 -5 years
   d. 5 -10 years
   e. 10 – 20 years
   f. > 20 years
3. How many years have you been contributing to open source (in any way)?
   a. < 1 year
   b. 1 -2 years
   c. 2 -5 years
   d. 5 -10 years
   e. 10 – 20 years
   f. > 20 years

Part II: Background on Ecosystem

1. Please choose ONE software ecosystem in which you frequently publish a package/library. If you have not published any packages/libraries, then pick an ecosystem whose packages/libraries you frequently use. Note: For the "Maven ecosystem", we are interested in the development of frameworks and libraries in Java, Scala, and other languages that share artifacts through Maven Central or other Maven repositories (for example through build systems or tools like gradle, sbt, ivy, or Maven itself).

   a. Bower

   b. Composer

   c. Maven

   d. Node.js/NPM

   e. NuGet

   f. Perl/CPAN

   g. PHP/Packagist

   h. Python/PyPi

   i. R/CRAN

   j. Other __________________

2. Which best describes your role in this ecosystem

   a. I am a core contributor

   b. I have submitted a patch or pull request

   c. I use packages/libraries of the ecosystem in my systems.

3. How many years have you been using the chosen ecosystem in any way?

   a. < 1 year

   b. 1 -2 years

   c. 2 -5 years

   d. 5 -10 years

   e. 10 – 20 years

   f. > 20 years

NB: If you identified yourself primarily as a publisher/developer of packages/libraries in the above section, please proceed to Part III. Otherwise, if you identified yourself primarily as a someone who reuses packages/libraries in the chosen ecosystem, please proceed to Part IV.

Part III: (Optional) Breaking Changes – Developer's Perspective

1. How often do you introduce breaking changes to packages/libraries you develop or contribute to?

   a. Never

   b. Less than once a year

   c. Several times a year

   d. Several times a month

   e. Several times a week

   f. Several times a day

2. How often do you face breaking changes from upstream dependencies?

   a. Never

   b. Less than once a year

   c. Several times a year

   d. Several times a month

   e. Several times a week

   f. Several times a day

3. What is your opinion on the following cost-sharing strategies for breaking changes (Strongly agree, Somewhat agree, Neither agree nor disagree, Somewhat disagree, Strongly disagree)

   a. Developers of components should invest extra work and effort to reduce impact of breaking changes on client applications

   b. Developers should make changes without caring about the amount of rework required for clients

   c. 3rd parties should take some of the burden reviewing changes, curating a selection of recommended libraries for clients, etc.

4. Which of these existing strategies do you (or the organization for which you work) adopt to delay/reduce the costs of braking changes for clients (multiple selections allowed)?

   a. Maintaining old interfaces (deprecation)

   b. Parallel releases

c. Release planning

d. Communication with users

e. Other __________________

5. How do you decide on an adoption strategy, and what measures (if any) are used when deciding on a strategy decision? (e.g. feedback from clients/users on proposed changes)

Part IV: (Optional) Breaking Changes – Client's Perspective

1. How do you declare the package/library versions that your project depends on?

    a. I specify an exact version number

    b. I specify a range of version numbers

    c. I specify only the name and always get the latest version

    d. Other __________________

2. Rank how these factors contribute to your decision when adding a dependency to your project. Assign a number from 1 to 8, with 1 being the highest ranked factor.

    a. The popularity of the package. ___

    b. How current the package is (latest release?) ___

    c. The quality of the package. ___

    d. The quality of the package contributors. ___

    e. The value added to your project by the dependency. ___

    f. The number of breaking changes in the dependency. ___

    g. The historical stability of the dependency (history of bugs, breaking changes, etc.) ___

    h. Other __________________: ___

3. How often do you face breaking changes from packages/libraries you use in your projects?

    a. Never

    b. Less than once a year

    c. Several times a year

    d. Several times a month

    e. Several times a week

    f. Several times a day

4. For most of the packages/libraries I reuse, I become aware of breaking changes by:

   a. reading about them on the dependency project's internal sources (not general public announcements)

   b. reading about them on the dependency projects external media (public announcements, social media)

   c. receiving a notification from a tool

   d. trying to build my project

   e. Other __________________

5. Have you ever been indirectly impacted by breaking changes caused by transitive dependencies?

   a. Yes

   b. No

6. If your answer to question 6 is Yes, can you briefly describe the experience: how did you detect and resolve the issue? __________________

7. What would be the most important feature you would like to have in a tool that detects possible direct impacts of breaking changes? __________________

8. What would be the most important feature you would like to have in a tool that detects possible indirect impacts of breaking changes? __________________