A Semantic Web Enabled Approach for Dependency Management

Ellis E. Eghan
Department of Computer Science and Information Technology, University of Cape Coast, Ghana
ellis.eghan@ucc.edu.gh
Juergen Rilling*
Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada,
juergen.rilling@concordia.ca

ABSTRACT

The use of external libraries in today's software projects allow developers to take advantage of features provided by such APIs without having to reinvent the wheel. However, APIs have also introduced new challenges to the Software Engineering community (e.g., API incompatibilities, software vulnerabilities, and license violations) that extend beyond traditional project boundaries and often involve different software artifacts. One potential solution to these challenges is to provide a technology-independent representation of software dependency semantics and its integration with knowledge from other software artifacts.

In our research, we take advantage of the Semantic Web and its technology stack to establish a unified knowledge representation of build and dependency repositories. Given this knowledge base, we can now extend and integrate other (heterogeneous) resources to allow for a flexible and comprehensive global impact analysis approach. To illustrate the applicability of our Semantic Web enabled modeling approach, we discuss two different applications. These applications illustrate how our modeling approach cannot only integrate and reuse knowledge from dependency management systems and other software artifacts, but also takes advantage of inference services provided by the Semantic Web to support novel software analytics services across artifact and project boundaries.

KEYWORDS

Semantic Web, knowledge modeling, dependency management, impact analysis

DECLARATIONS

Funding: Discovery Grant - NSERC Canada
Conflicts of interest/Competing interests: N/A
Availability of data and material: N/A
Code availability: N/A

1 Introduction

Traditional software development processes, focus on closed architectures and platform-dependent software, restricted potential code reuse across project, and organizational boundaries. With the introduction of the Internet, these boundaries 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 have become an essential part of today's development practice [2]. Now, Software developers can take advantage of features provided by external libraries through their Application Programming Interfaces (APIs) without having to reinvent the wheel [3], [4].

Automated dependency management environments have been introduced to simplify and automate the integration, management, and reuse of external libraries during development. Build systems and dependency management tools automatically download and manage all necessary dependent components (including transitive dependencies), update

dependencies to their latest versions, and perform dependency mediation (conflict resolution) when multiple versions of a dependency are encountered. Among the most widely used open-source build (dependency) management environments are Maven Central[1], npm[2], and RubyGems[3].

Existing research has shown how mining knowledge captured in 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].

While current 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 global software engineering challenges. Such analysis and modeling techniques must be able to capture not only intra project dependencies but also inter project dependencies across project boundaries as well as within complete software ecosystems. Therefore, a technology-independent representation of software dependency semantics is needed that can provide the ability to seamlessly integrate knowledge from other software artifacts and support dependency analysis that can reason upon knowledge that is explicit and implicit captured in such a knowledge base.

Knowledge graphs are recognized by many industries as an efficient approach to data governance and data integration. A central promise of knowledge graphs is that heterogeneous data, i.e., data from unstructured data sources up to highly structured data, can be harmonized and linked so that the data of higher quality can be used for subsequent tasks such as machine learning. While many domains (e.g., enterprises [29]) have started to make knowledge graphs an integrated part of their solution space, the software engineering community has yet to embrace this move.

While there are many works on creating software analytics tools or services to support software analysis tasks, their applicability is often limited for several reasons; First, these tools have remained information silos by relying on their own proprietary data collection and data models. Whilst these models work well for supporting tool specific analytics services, they limit their ability to share, reuse, and integrate data and analysis results with other software analytics tools and knowledge resources. Second, the underlying knowledge models used by current analysis approaches lack support for a seamless integration of new knowledge resources or the ability to deal with incomplete data. Thirdly, their analysis support is often limited by the underlying knowledge model and provide limited flexibility in terms of supporting user specific analysis needs.

In our research, we take advantage of the Semantic Web (SW) and its technology stack (e.g., ontologies, Linked Data, reasoning services) **to establish a unified knowledge graph representation of build and dependency repositories**. Based on this knowledge graph, we can now extend and integrate this knowledge with other (heterogeneous) resources to allow for a **flexible and comprehensive global impact analysis** approach that provides library producers as well as consumers with new insights to guide them during the evolution of their libraries.

The research in this paper is a continuation of our previous work on semantic modeling in which we introduced a Security Vulnerability Analysis Framework (SV-AF) that establishes traceability links between the National Vulnerability Database (NVD)[4], Maven dependency repository, and the source code of projects [9], [10]. In this paper, we describe in detail our design rational and the process we applied to create knowledge graph representation of build and dependency semantics.

Our research is significant for several reasons:

1) We introduce a formal Software Build System Ontology (SBSON), which captures concepts and properties for software build and dependency management systems that allows for a **standardized representation** of the semantics of these systems. This unified representation allows for the introduction of **new types of dependency analysis** at the level of individual and across different build management systems.

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

2) We illustrate how SBSON can be further integrated with other knowledge resources, by taking advantage of the Semantic Web and its ability to model and s**eamless integrate** heterogeneous knowledge resources.
3) We describe several key design decisions we applied to enhance the ontology design and illustrate how **our approach takes advantage of reasoning services** provided by the Semantic Web to further enrich our knowledge representation.
4) We demonstrate how our integrated knowledge modeling approach can support **novel types of knowledge driven software analytics services** that are flexible (user defined queries) **and** take advantage of SW inference services.

The remainder of the paper is organized as follows: Section 2 provides an illustrative example to motivate the significance of this work. Section 3 provides background related to dependency management and semantic web technologies. Section 4 presents the core of this paper, SBSON. Section 5 demonstrates two different applications of SBSON. Related work and potential threats to validity are discussed in Sections 6 and 7, respectively. Finally, Section 8 offers concluding remarks and outlines future research directions.

2 Motivation

Although the reuse of third-party libraries provides developers with productivity gains by not having to re-implement already existing functionality, this form of code reuse also introduces new technical and organizational challenges [11]. Some of these challenges identified and discussed in existing research are:

- selecting the most relevant library among several alternatives [12], [13], [55],
- how to use features provided by these libraries [13], [14], [56]
- 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], {57]

To address these challenges, existing approaches analyze knowledge found within repositories such as dependency management repositories (e.g., Maven Central, npm), source code repositories (e.g., GitHub[5]), vulnerability databases (e.g., NVD), and Q&A forums (e.g., StackOverflow[6]). However, as mentioned in the introduction, most of these approaches treat these repositories as **information silos, caused by**: (a) a lack of standardized knowledge representations across repositories, b.) analysis pipelines that are typically proprietary to the specific knowledge resource (repository), and (c) resources and analysis results that are not easily sharable among knowledge resources.

We argue that most software analytics tools have remained in information and application silos due to their lack of Findability, Accessibility, Interoperability and Reusability (FAIR) of data and analysis results. These information and application silos make it (1) extremely costly and difficult to extract data and (2) use it for anything other than its original purpose.

The first problem stems from the governance of data. While data silo owners can maintain full control over their data and establish their own governance processes, data silos contradict the **FAIR** data principles, limiting the ability to identify and find individual facts and data elements across resource boundaries. Data should be accessible and interpretable by both humans and machines and be stored in trusted repositories. Such trusted repositories should support data consistency checking and provide global access, with data and analysis results being identifiable and accessible **across** tool, project, and knowledge resource boundaries. Key in transforming such data silo into information hubs is a formal, accessible, shared knowledge representation language.

The second problem follows the lack of support for the FAIR data principle, with data silos that cannot efficiently handle the full range of contexts that are potentially available, since there is typically one data silo per application.

[5] https://github.com/
[6] https://stackoverflow.com/

Such a*pplication thinking*, results in software applications and associated data structures that are optimized for a specific purpose at a certain point in time. Efficient data exchange is rarely a primary concern therefore proprietary data models are used. Instead of placing data and larger analytics workflows at the center of their system design, applications often continue to be lined up and optimized separately. One must consider a more global perspective, allowing for software analytics approaches to go beyond application specific thinking, and support the inference of new knowledge across resource boundaries.

The following scenarios illustrate how such an integrated knowledge modeling approach allows for the **integration of heterogenous knowledge and** the support of **new, user defined types of knowledge-based software analytics services**.

Scenario #1: Support for bi-directional dependency analysis. Current build tools provide support for automatic dependency management; a project only needs to specify the third-party libraries it directly depends on, and the build tool will automatically include all required transitive dependent components. However, as shown in Figure 1(a), such dependency analysis only supports unidirectional dependencies. While unidirectional dependency models work well for managing build dependencies at the individual projects level, they lack the necessary expressiveness to let users take full advantage of the stored dependency information.

For example, Maven's native support for **impact analysis** allows a developer to identify all components **a particular project depends on**. As illustrated in Fig. 1(a), a component C might depend on components D and E. However, Maven's dependency model **makes it impossible for an API (library) producer to identify which other projects (clients) depend (either directly or indirectly) on his API**. Such additional dependency information is useful when a developer wants to determine the potential impact of a change to his library on other libraries/projects within the current build management ecosystem. Furthermore, given that each dependency system relies on its own dependency management ecosystem, these systems do not support dependency analysis **across different dependency** management ecosystems.

Using SW (and its supporting technology stack), we mine and model dependencies across projects and even build management ecosystems to create a **"global" bi-directional dependency graph** (Figure 1(b)). In this enriched bi-directional knowledge model, **library producers can now identify all components (consumers) which depend (directly or indirectly) on their libraries**. For example, the developer (producer) of component C can now identify that components A, F, and G (clients) can potentially be impacted by a change to C.

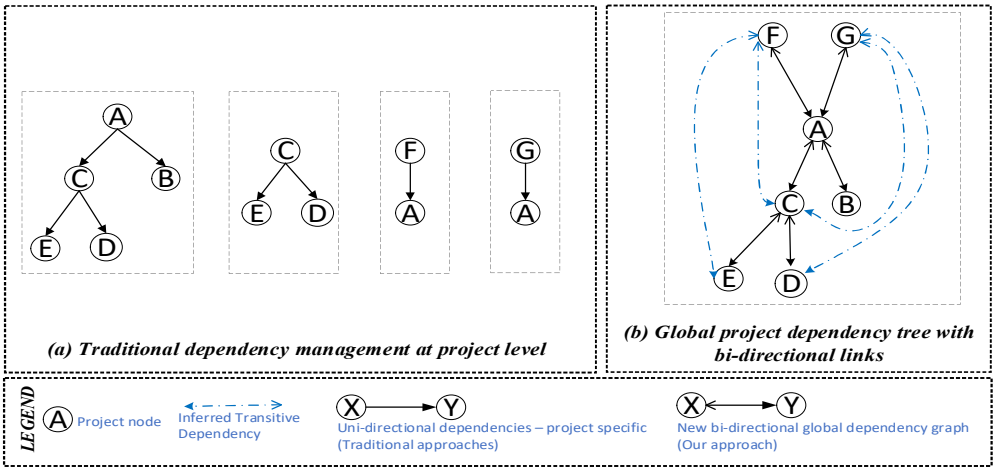

Figure 1: Overview of motivating scenario #1

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 will require access to source code, license files, and vulnerability databases. While existing approaches

and tools support such types of analysis using project dependencies (e.g., VersionEye[7], SourceClear[8], OWASP-DC[9]), their current analysis support remains limited by their knowledge representation (e.g., support only for unidirectional dependencies) as well as a lack of extensibility of their knowledge model and therefore a lack of seamless integration of other (new) knowledge resources in their analysis.

In contrast, our approach takes advantage of the SW to establish traceability through a **global project knowledge graph**. This global dependency graph allows us to a.) integrate concepts and facts from other software knowledge repositories, while b.) supporting the inference of new knowledge, and c.) allowing analysis results to become an integrated part of the knowledge model.

For example, in Figure 2 a traceability link is established **between** project E instances in the **vulnerability** and **dependency ontology**. Using this traceability link, we can now **infer** that projects C, A, F, and G can potentially be affected by a vulnerable in project E, due to their (transitive) dependency on project E. In addition, having access to the license ontology, we can now identify that project A may introduce a license violation. For example, given the transitive dependency between project A and D, and project D having a license which is conflicting with the license in A.

In addition, these two scenarios illustrate how our approach takes advantage of the **Open World Assumption**[10] (OWA) which must hold when modeling and analyzing these software engineering resources to be able to deal safely with incomplete data. Meaning, the lack of information cannot be used to infer further knowledge, unlike most existing source code analysis approaches which are based on the closed world assumption (CWA) [24]. For example, in Figure 2, we do not have any established traceability link between project F's instance in the dependency model and the vulnerability model. This does not mean project F has no security vulnerabilities; we cannot infer that fact at the moment. Using the SW, we can safely deal with such incomplete data, support incremental knowledge population and take advantage of its inference services [25], [26].

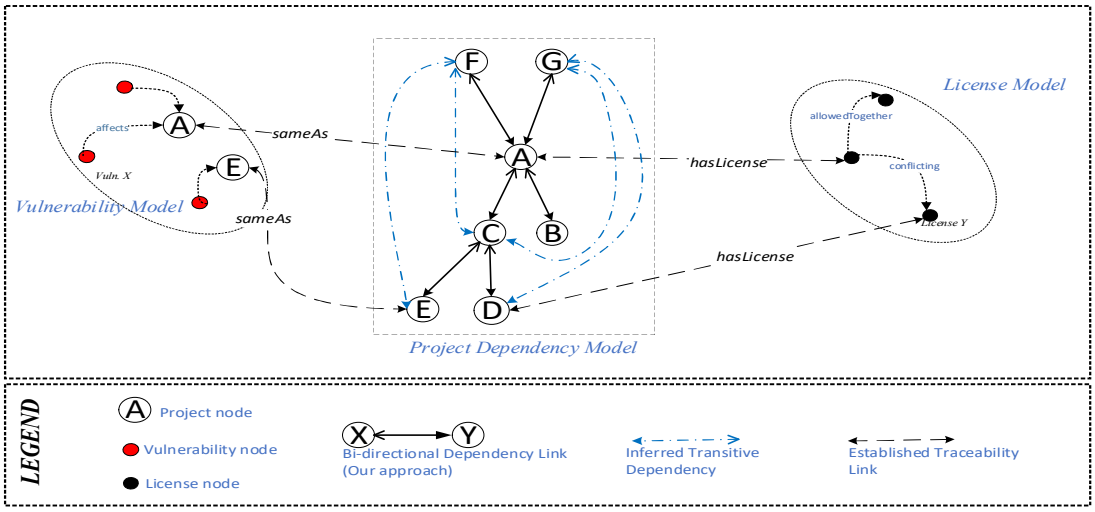

Figure 2: Motivating scenario #2 - Integrating build information and knowledge from heterogeneous software repositories

3 Background

3.1 Build Systems and Dependency Management

Build systems transform the source code of a software system into deliverables. Despite their different design paradigms [27], 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

[7] https://www.versioneye.com/
[8] https://www.sourceclear.com/
[9] https://jeremylong.github.io/DependencyCheck/
[10] https://en.wikipedia.org/wiki/Open-world_assumption

build dependencies. Dependency Management is one of the critical features of existing build automation tools. Build automation tools use project dependency definitions captured in specialized build repositories (e.g., Maven Central, npm, PyPi[11], and RubyGems) to provide dependency management features such as transitive dependencies and dependency mediation (conflict resolution).

Transitive dependencies: If A depends on B, which in turn depends on C, then C is considered to be transitively dependent on A. Part of Maven's appeal is that it can manage these transitive dependencies and shield developers from having to keep track of all build dependencies required to compile and run an application [28].

Dependency mediation: In Java, the JVM is unable to differentiate between multiple versions of an API and will always choose the first occurrence of an API in a project's class-path, independent of its version. In cases where multiple versions of a dependency are encountered, Maven attempts to resolve such version conflicts by using only the version of the dependency closest to the root of the dependency tree. However, his type of conflict mediation can lead to potential runtime failures, which are not identified during the build or compilation process.

3.2 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" [29]. 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, the 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 formal descriptions 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 in addition to 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.

Ontologies in Software Engineering. Representing software in terms of knowledge rather than data, ontologies provide a better support for representing the semantics of software [29] compared to relational databases and other data representations based on the CWA, as sharing and extending of schemata are not natively supported. In contrast, Semantic Web knowledge models are extensible, allowing the addition of new knowledge without affecting existing knowledge. Unlike relational databases or other data representation based on the CWA, extending an existing 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). Additional benefits identified by [30] are the Semantic Web which 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.

[11] https://pypi.org/

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

4.1 Overview

Our approach is based on a semantic knowledge model using ontologies. Ontologies not only allows us to provide a standardized knowledge representation for software build and dependency knowledge, but also to integrate this knowledge with other software artifacts to support novel knowledge-driven dependency analysis services.

The proposed model adheres to the following design criteria proposed by [31], [32]:

- **Unambiguous Semantics.** The primary motivation for using ontologies over other modeling approaches is to enrich information with a formal semantic representation. 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 paper 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. Using the RDFS++ subset of OWL2 inference allows us to maintain the scalability and tractability of our model [31].

In the next section, we explain the knowledge engineering methodology which we applied for the construction of our unified knowledge model and the design decisions we made to address some of the open research challenges discussed in our research motivation (Section 2).

4.2 Knowledge Modeling and Engineering

Different knowledge engineering methodologies have been discussed in the literature (e.g., Noy et al. [33], Van der Vet et al. [34], and Uschold et al. [35]. Noy et al. [33]), in their knowledge-engineering approach for ontology development, proposed the following seven main 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. [34] proposed a bottom-up approach for building ontologies. Their approach depends on atomism, that is, objects are composed of indivisible units called “atoms.” They use part-whole relations to group basic concepts into “superconcepts”.

Our methodology consists of five major steps (Figure 3) which are similar to the methodology introduced by Noy et al. [33]. In Step (1), we perform a manual review of the documentation from selected build and dependency management systems and their repository structures to identify and extract their concepts and properties. In Step (2), we manually inspect these extracted concepts and properties for each build system to derive initial versions of our system-specific ontologies. During Step (3), we use a bottom-up approach (similar to the knowledge modeling approach presented by Van der Vet et al. [34]) to identify and move shared concepts and attributes from these system-specific ontologies into different layers of abstraction (upper ontologies). We then further refine and enrich these ontologies, by adding relations and properties, to have a model that is semantic rich enough to allow for the inference of knowledge using basic SW reasoning (RDFS++). In Step (4) we populate our knowledge model with facts from projects published in open-source build repositories. The evolution of our ontologies with new build and dependency management systems and concepts as they become available is part of Step (5).

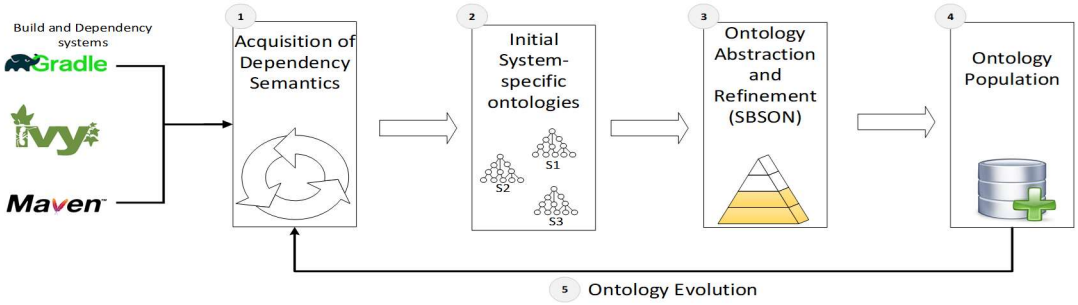

Figure 3: An overview of our knowledge modeling methodology.

The outcome of our 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 similar to the modeling approach introduced by the Object Management Group (OMG)[12], where the top layer captures core elements, which are extended and refined throughout the abstraction hierarchy. Figure 4 presents an overview of the different ontology abstraction layers in SBSON. For a complete description of these ontologies, we refer the reader to [36].

Within our knowledge hierarchy, the *General* Concepts layer captures the omnipresent core concepts related to software evolution. The *Domain-Spanning* Concepts extents the *General* Concepts layer by capturing concepts (e.g., measurements) that span across several subdomains (e.g., vulnerability databases, version control systems, and source code). Some *Domain-Spanning* concepts found within this layer are introduced in Section 5 when other SE knowledge sources are integrated with SBSON. Concepts in the *Domain-Specific* layer are common to resources in a domain, such as software build and dependency concepts. Finally, the *System-Specific* layer represents knowledge (concepts and properties) that is specific to a given data source or system and not commonly shared across the domain. In what follows, we describe in detail the five knowledge modeling steps we applied.

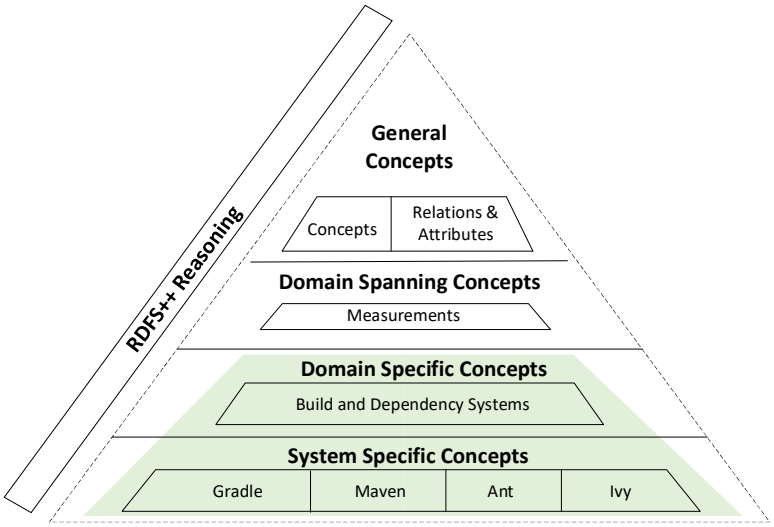

Figure 4: An overview of the different ontology abstraction layers in SBSON.

4.2.1 Step (1): Acquisition of Dependency Semantics

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

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

system. Table 1 provides an overview of these three systems and some general statistics of the Maven Central repository are provided in Table 2. It should be noted that, although we only studied systems that use the Maven Central repository, our knowledge modelling approach can easily be extended to different build systems and repositories (see Section 4.2.5.)

Table 1: Overview of the 3 build and dependency management systems used in our case studies

Name	Maintainer	Default repository	Dependency management features				
			Transitivity	Filtering	Version Ranges	Scope	Default Resolution
Ivy (with Ant)	Apache	Maven Central	YES	YES	YES	NO	Latest version
Gradle	Gradle		YES	YES	YES	YES	Latest version
Maven	Apache		YES	YES	YES	YES	Nearest

Table 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 (Table 1) support dependency management features such as transitivity, dependency filters (exclusions), and version ranges, only Maven) and Gradle support dependency scopes that allow to limit the transitivity of a dependency used for various build tasks. 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, Ivy and Gradle choose (by default) during version conflict resolution always the latest version of a dependency, while Maven 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 5 provides an overview of the three system-specific ontologies we extracted.

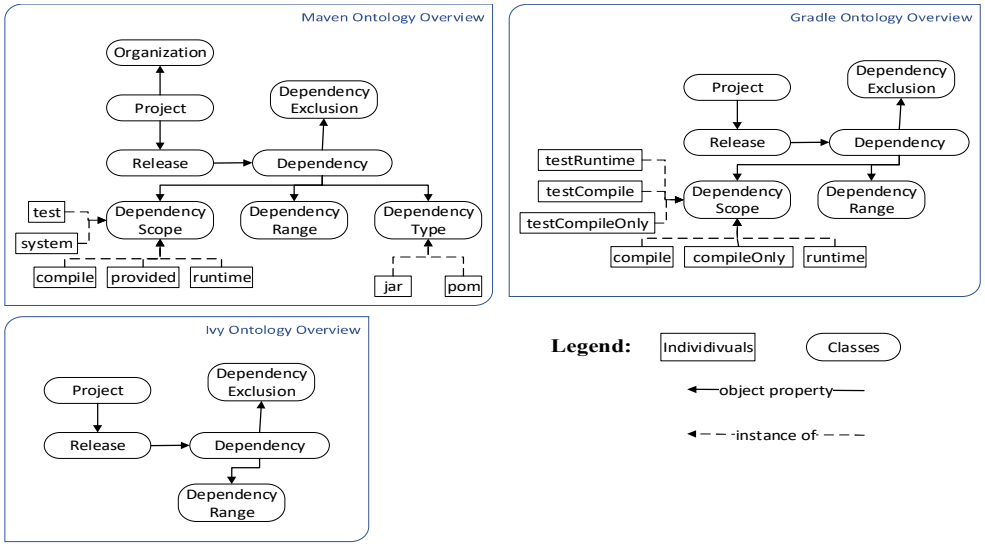

Figure 5: Overview of individual system-specific ontologies for the analyzed systems.

A main challenge we had to deal with during this modeling step was to identify and resolve differences in syntax and structure among concepts and properties in the three systems. Table 3 and Figure 6 show examples of such representation differences for capturing open and half-open intervals[13] ranges and dependency definitions (e.g., Ivy uses "]" to declare an open minimum version while Maven uses "(").

Table 3: Comparison of how different dependency management systems handle 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

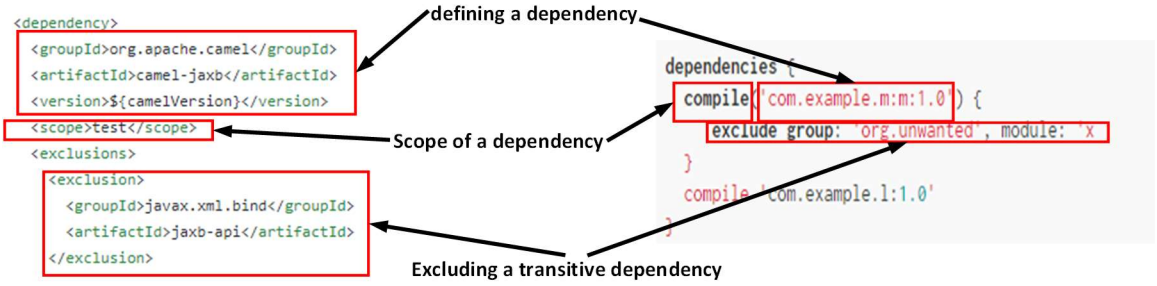

Figure 6: Example of syntax and structural differences between Maven (left) and Gradle (right) dependency definitions

4.2.3 Step (3): Ontology Abstraction and Refinement

In this step, we use the extracted system-specific ontologies to abstract a software build-dependency domain ontology. This *Domain-specific* layer not only promotes reuse of concepts shared across system level ontologies, but also improves the traceability among system level ontologies by unifying the overall knowledge representation. It also allows for the linking of system-level ontologies via abstracted shared concepts and properties found in the domain ontology. More specifically, in this step of our methodology, we identify 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.

Although the identification of shared concepts can be considered mostly a straightforward task, providing a design that allows for the inference of new knowledge is challenging. Such advanced design requires to establish traceability links between domain and system-level ontologies that can support at the application level new types of API dependency analysis. In what follows, we describe in detail how we enrich our 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, to improve the readability, we use prefixes as substitutes to the fully qualified names of our ontologies. The ontology prefixes used in this paper can be dereferenced using the URIs shown in Table 4.

[13] 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.

Table 4: Ontology Prefixes

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

4.2.3.1 Modeling Dependency Links

Problem. As shown in Figure 7(a), a defined dependency between any two project releases can have additional characteristics associated, such as the version range of the dependency as well as 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[14]. In the following, we illustrate the use of the property reification pattern to model facts about the dependency relation between two project releases.

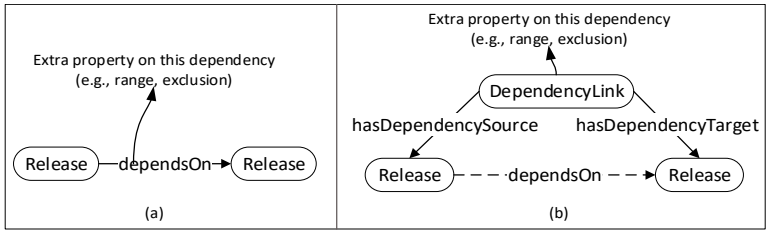

Figure 7: An illustration of (a) generic dependency between two releases, and (b) how property reification pattern is adopted in modeling dependency links

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 7(b). This reification design 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.

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

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*.

Solution. We address this challenge by reusing an existing OrderedList Ontology design pattern[15] to model projects and the order of their releases. The OrderedList ontology, illustrated in Figure 8(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 8(b) illustrates our extension of the OrderedList ontology, which now assigns one ordered list to each project. Multiple releases of a project are subsequently ordered by assigning them as items to slots of the project's ordered list. Figure 8(c) shows an example of a project with three ordered releases.

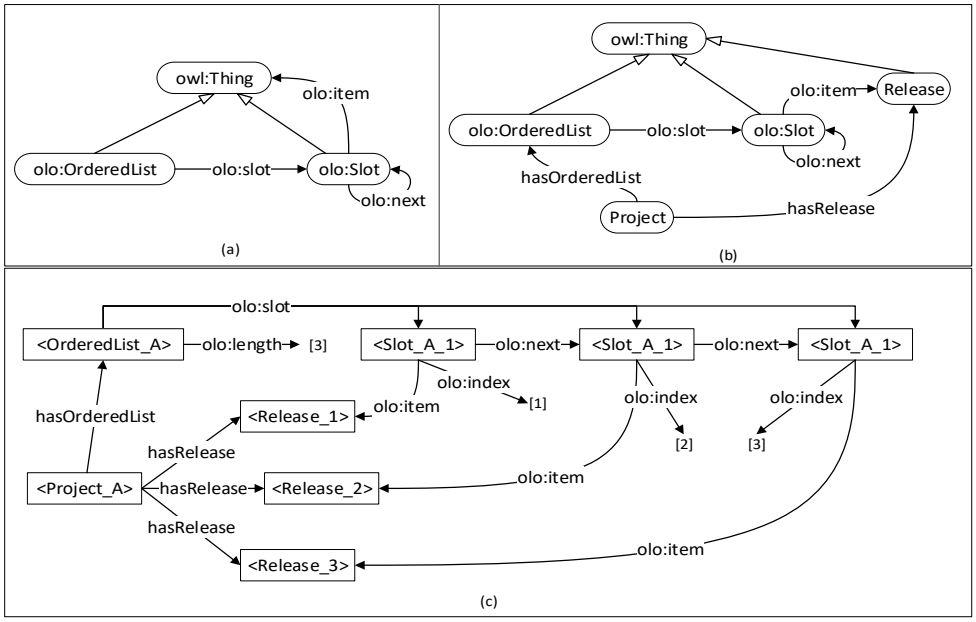

Figure 8: (a) The OrderedList Ontology, (b) how we model the order of project releases with the OrderedList Ontology, and (c) an illustrative example of a project and its ordered releases.

4.2.3.3 Modeling Version Ranges

Problem. Manually upgrading dependencies is a tedious and error prone 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 discussed in Section 4.2.2, build and dependency management systems use version ranges with different syntaxes.

Solution. Figure 9 shows the integration of concepts from Figures 7(b) and 8(b) to create an effective and flexible model to represent dependency version ranges (at the 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 a dependency version is inferred are provide in Section 4.2.3.5.

[15] http://smiy.sourceforge.net/olo/spec/orderedlistontology.html

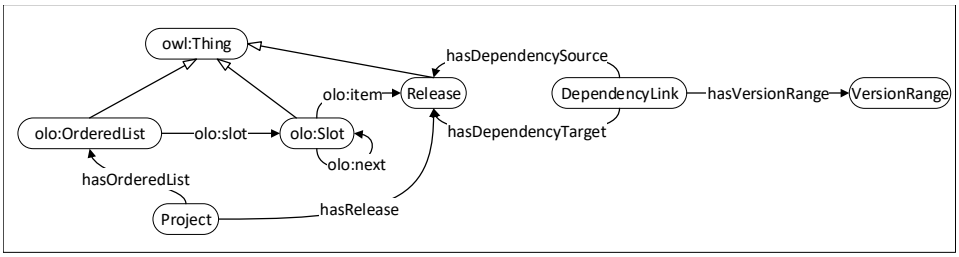

Figure 9: Concepts used to model and reason about dependency version ranges

4.2.3.4 Modeling Dependency Exclusions (Filtering)

Problem. Dependency exclusion is a feature provided by many dependency management tools (e.g., Maven, Gradle) to support dependency mediation. It allows 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 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 only per-dependency exclusion. While our approach currently only supports per-dependency exclusions it can easily be extended to support per configuration/module dependencies.

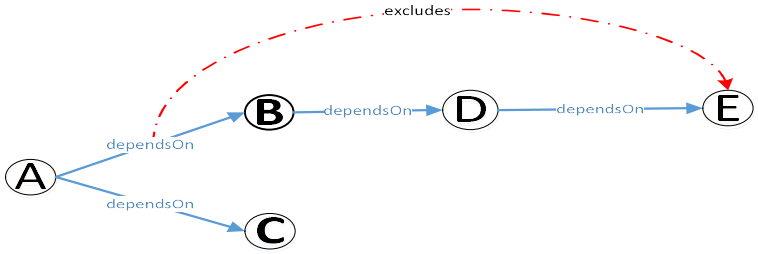

Figure 10: Transitive exclusion at per-dependency scope

Figure 10 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 querying for all transitive dependencies of 'A', the result should be {B, C, and D}. Since exclusions are dependency specific, the query results will 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. Similar to the dependency version ranges, we use again the <sbson:DependencyLink> concept from the property reification pattern (see Figure 7(b)) to define any dependency-level exclusions on projects or releases through the <sbson:excludesProject> and <sbson:excludesRelease> properties (Figure 11).

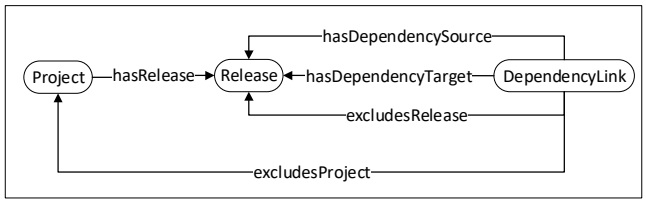

Figure 11: Concepts used to model and reason about dependency exclusion

In what follows, we describe how we can now, using the dependency version ranges and exclusions design, infer bi-directional direct and transitive dependencies.

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 and the triplestore to reason about 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 about direct dependencies based on version ranges, and transitive dependencies.

Direct Dependency Reasoning. To allow for the automatic resolution of direct dependencies, we define three (3) rules which infer the correct instance of a dependency version, will be assigned to the range of the <sbson:hasDirectDependencyOn> property. The rules are based on the following version ranges: exact versions (Figure 12), versions lower than a specified value (Figure 13), and versions greater than a specified value (Figure 14). The rules take advantage of the ordered list pattern (see Figure 8(b)) and the dependency link reification pattern (see Figure 7(b)) to allow for the inference of the final dependency version to be used.

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

Figure 12: 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 13: 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 14: Inferring DirectDependencyOn based on a "greater than" version range

Transitive Dependency Reasoning. This reasoning provides flexible and scalable inference of transitive dependencies in the absence or presence of dependency exclusions. A pre-computation can be performed and inferred triples can be materialized in the triple store so that future queries run more efficiently. Since SWRL does not allow for Negation as Failure[16], rules such as Person(?p) ^ ¬ 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 infer 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 for a given project dependency. Using this property, our rules (Figures 15 and 16) can infer now transitive dependencies in both the presence and absence of exclusions.

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

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

Figure 15: 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 16: Inferring hasTransitiveDependencyOn in the presence of exclusions

4.2.3.6 A Unified Knowledge Representation

The result of our modeling process is SBSON, which captures knowledge from build and dependency management systems at different abstraction levels. Figure 17 provides an overview of the core SBSON concepts and object properties. It should be noted that data properties have been omitted to improve the readability of the figure.

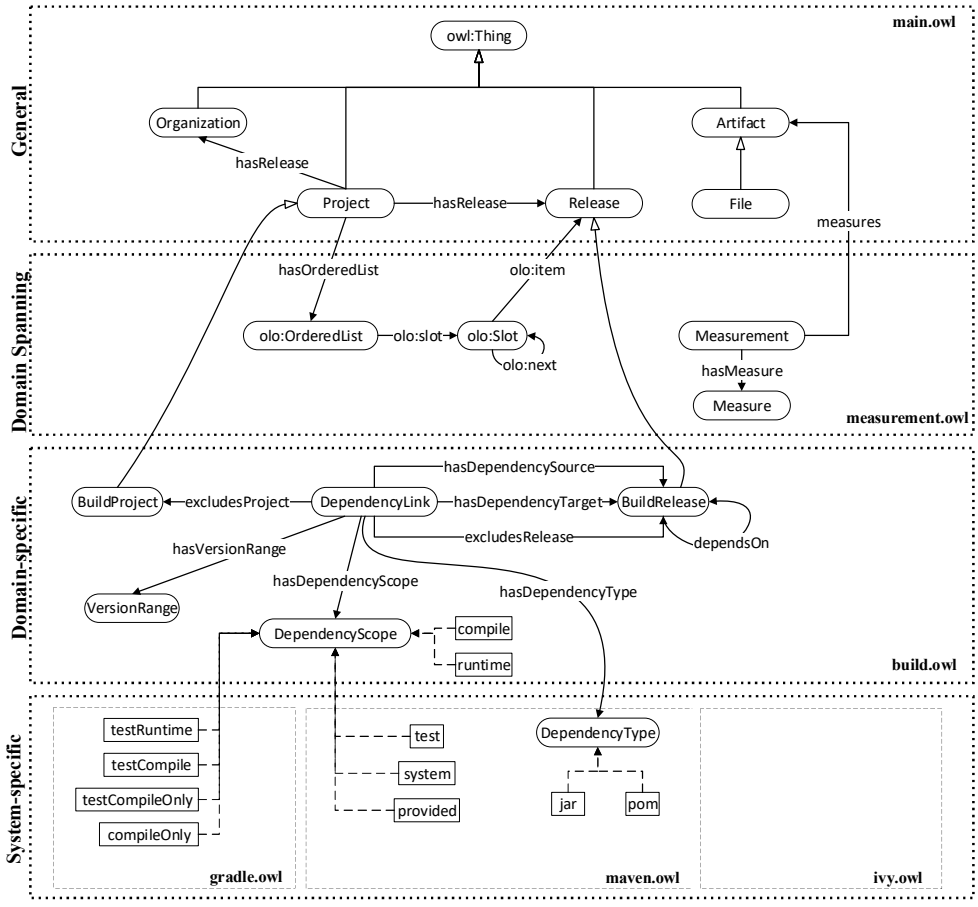

Figure 17: Overview of the concepts and (object) properties in the unified SBSON family of ontologies

A key concept in our knowledge model is the <sbson:BuildRelease> (*Domain-Specific* layer), 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* layer). 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* layer). In our modeling approach, build releases define their dependencies on other releases using a <sbson:DependencyLink> (*Domain-Specific* layer). 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-Specific* layer. Scope of dependencies, as well as dependency types are specific to an individual build system and are therefore modeled as part of the system ontologies.

4.2.4 Step (4): Ontology Population

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

Figure 18 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).

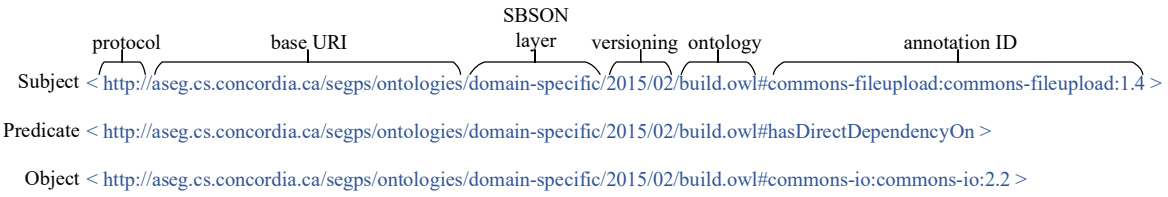

Figure 18: Anatomy of the URI of a generated triple

4.2.5 Step (5): Ontology Evolution

The last step of our methodology reflects that our knowledge modelling approach is an iterative process, with ontologies evolving as additional build and dependency management systems is added to the knowledge model. The addition of new system-specific ontologies can lead to changes in the domain ontology. In addition to the inclusion or promotion of concepts and properties to the domain ontology capturing common dependency management features and semantics, there is also the possibility that existing domain concepts and properties will be demoted to the level of system-specific ontologies. As discussed earlier, a key benefit of using ontologies is that they can be extended as relationships and concept matching are easy to add to existing ontologies without impacting dependent processes and analysis services. However, a knowledge engineer will face the challenge to establish an equilibrium between the amount of information needed and the granularity of the knowledge available to produce useful results.

5 Example Applications Supported by SBSON

In what follows, we introduce two examples, to illustrate the flexibility of our modeling approach in being able to integrate knowledge resources and making this knowledge accessible and reusable across resource boundaries through user defined queries that take advantage of the Semantic Web and its inference services. The first application scenario demonstrates how SBSON can support impact analysis to identify potentially affected components due to API breaking changes. For the second example, we illustrate how the dependency information modeled in SBSON when combined with bi-directional links to vulnerability information resources, can be used to provide a global vulnerability analysis that can identify all projects which directly and indirectly depend on a known vulnerable component.

5.1 Early Detection of API Breaking Change Impacts

Objective: As discussed 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 selects the version of the dependency closest to the root of the dependency tree. However, such conflict mediation, can lead to potential runtime failures that are not identified during the build or compilation process.

In what follows we illustrate, how our approach can support API consumers in identifying potential impacts of an API change on their product. Our approach takes advantage of our knowledge model that supports the analysis of both direct and indirect (transitive) third-party library usage across thousands of open-source projects.

Approach: Since part of this API impact analysis requires access to source code information, we introduced our SOCON ontology which is an extension of SEON's domain-level source code ontology [30]. SOCON introduces additional concepts and properties to model knowledge relevant to 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. Figure 19 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.

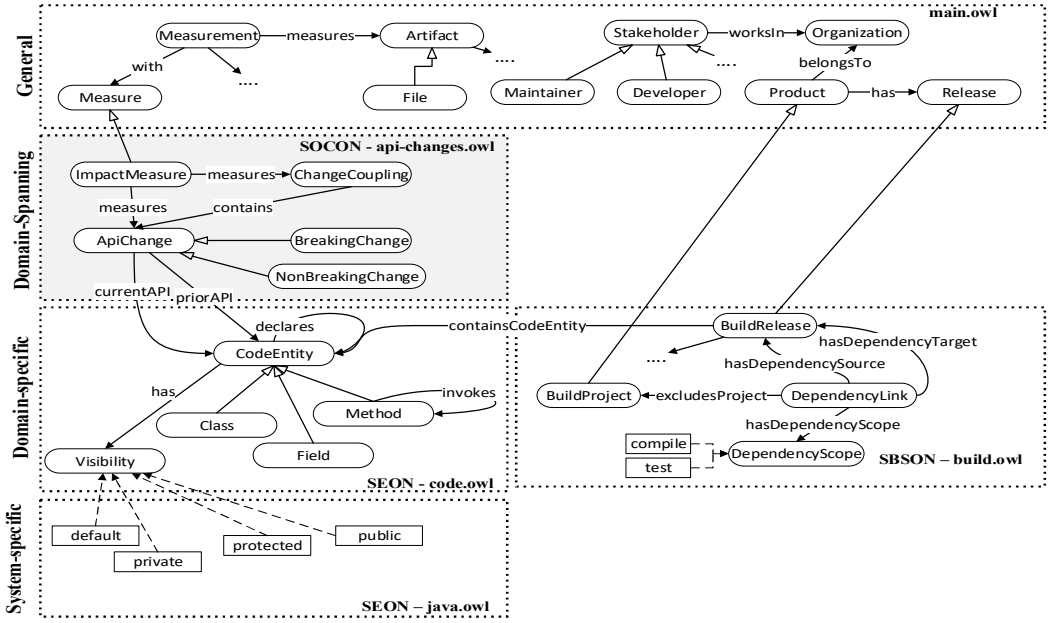

Figure 19: Ontologies and concepts involved in API change impact analysis

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 granularity 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, 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. Given this unified representation, developers can now use (user and predefined) SPARQL queries to analyze whether their application is potentially exposed to direct and indirect breaking changes. A complete description of our ontologies can be found at [36].

Results: In what follows, we report our results from a case study, which we conducted on ASM[17], a Java bytecode manipulation library which underwent a radical redesign from release 3.X to 4.0. As part of its redesign, release 4.0 introduced several breaking changes (e.g., interfaces were changed to abstract classes, breaking previous 3.X API versions). Our case study analyzed the impact of these breaking changes to the dependent projects in the Maven ecosystem. Table 5 summarize the details of our ASM datasets.

Table 5: Summary of ASM releases included in the case studies

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

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

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. 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.

The query in Figure 20 identifies all projects that are dependent (either direct or transitive) on different versions of the ASM library. The query in Figure 21 (an extension of Figure 20) 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.

```
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 20: 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 21: SPARQL query to identify transitive usage of API elements impacted by breaking changes

The boxplots in Figure 22 summarize the distribution of dependents among clients as well as the usage of potential incompatible APIs within client and dependent projects. Clients, on average, include 5 dependents which may introduce different versions of the ASM library as part of their classpath. Further analysis (Figure 22) shows that on average 0.21 of the ASM API 4.X interface and 0.34 of the interface from earlier ASM version (ASM 3.X or earlier) are invoked by a dependent.

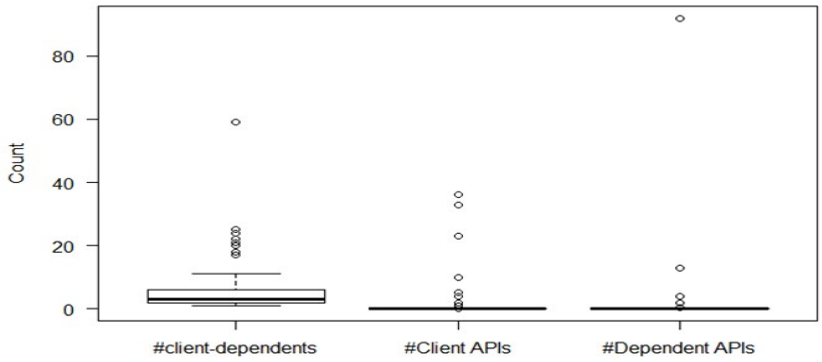

Figure 22: Distribution of client dependencies and their usage of incompatible ASM APIs

Table 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 used. In this example, an unexpected runtime exception will be thrown when the fromExpression method, since it indirectly invokes the now excluded ASMv3.1 ClassVisitor and MethodVisitor APIs

Table 6: Some client projects potentially impacted by transitive incompatible ASM API versions

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)

5.2 SV-AF: Security Vulnerability Analysis Framework

Objective: It is generally accepted that inadvertent programming mistakes can lead to software security vulnerabilities and attacks [37]. Mitigating such 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.

Existing work (e.g., [38]) attempts to minimize the introduction and exploitation of software security vulnerabilities. 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., NVD) 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 our previous work [9], we introduced SV-AF, which establishes traceability links between security and software databases for automatically tracing source code vulnerabilities at the API level across project boundaries. In the following example, we show, how our unified knowledge representation, with its bi-directional linking of other knowledge resources can be used to a.) analyze the potential impact of a vulnerable component on a software ecosystem and b.) identify vulnerable components a system might directly or indirectly depend on. More specifically, how our SBSON ontology in combination with the SEON and SEVONT ontologies can support novel types of vulnerability analysis at a global scope.

Approach: 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. SV-AF uses the Probabilistic Soft Logic (PSL) framework [39] to establish weighted links between ontological models of vulnerability databases (SEVONT) and software dependency repositories (SBSON). These traceability links are created based on semantically identical or similar concepts within the different knowledge sources. Similarities among SEVONT-SBSON instance pairs are determined based on literal information such as name, version and vendor. Using user defined rules, the PSL framework computes similarity weights between all possible instance pairs in the knowledge base (total of |SEVONT| x |SBSON| instance pairs). These computed similarity weights, based on a given similarity threshold, are used to infer owl:sameAs relations between similar instances found in the two ontologies. The owl:sameAs construct is a built-in OWL predicate used to align two concepts from different ontologies. More details on the ontologies, ontology alignment process, and evaluation of the SEVONT-SBSON alignment can be found in our existing work [9].

The result of this alignment processes is a unified that integrates 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 23 (data properties have been omitted to improve readability of the figure).

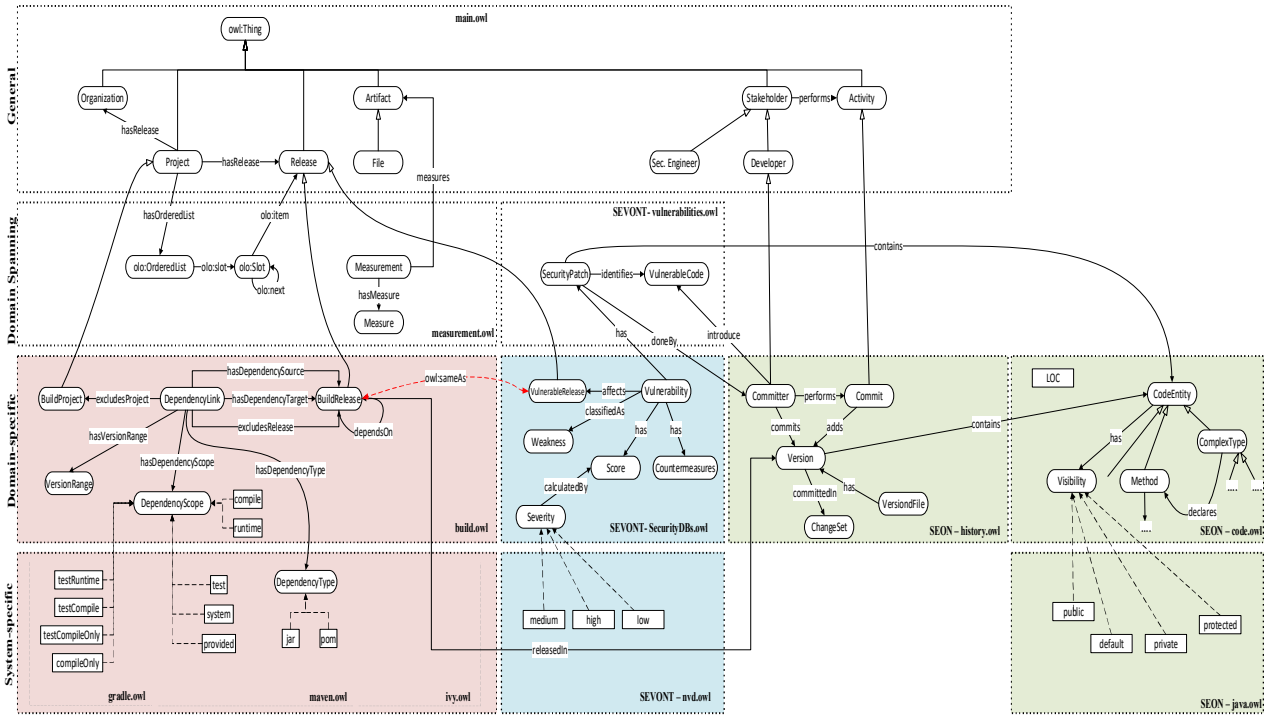

Figure 23: The SV-AF ontology concepts involved in API-level vulnerability impact analysis

Results: We performed a case study using the NVD and Maven Central repositories. For the case study, we downloaded the Maven repository (Table 7) and all NVD vulnerability xml feeds from 1990 to 2016. The dataset includes 74,945 unique vulnerabilities that affect 109,212 unique software products. Our study showed that 750 (or 0.062%) of all Maven projects contain known security vulnerabilities already reported in the NVD database. Further analysis revealed that many projects not only suffer from one but from multiple vulnerabilities. We also 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. This information about potential vulnerability components 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.

Using the bi-directional links in our knowledge 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 provide a more holistic analysis, which allows us to identify all projects which **directly and indirectly depend on a given vulnerable component**. Figure 24 illustrates a typical usage scenario for our modeling approach. While the Geronimo-jetty6-javaee5 (version 2.1.1) 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 vulnerabilities), making also Geronimo-jetty6-javaee5 potentially a very vulnerable component.

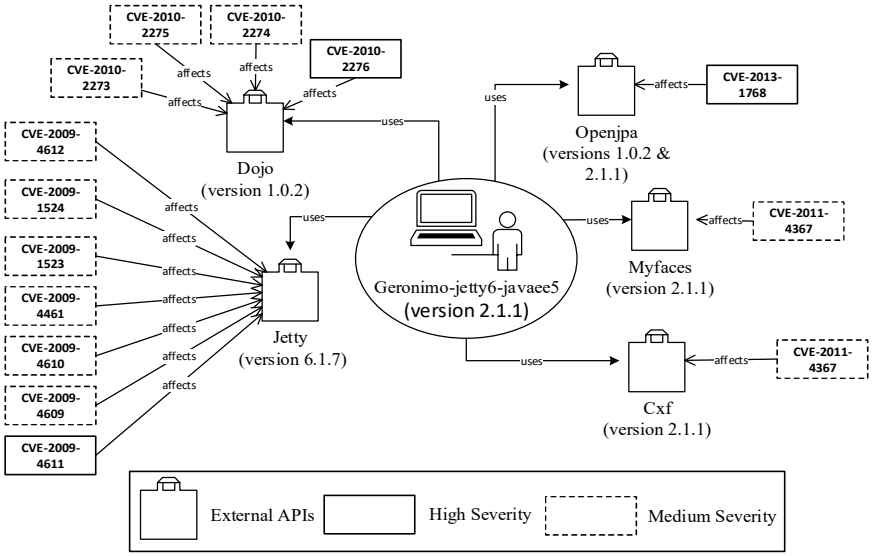

Figure 24: Geronimo-jetty6-javaee5 using 5 vulnerable projects (level 1 dependencies)

In addition, we take advantage of **transitive and subsumption inferences** applied at the source code level to identify vulnerable APIs and trace their impact to external dependencies (details of our source code transitivity and subsumption inferencing can be found in [9]).

Using our SEVONT, SEON, and SBSON ontologies, we can now execute the SPARQL query in Figure 25 to restrict the scope of our transitive dependency analysis by including only those components that have an actual call dependency to the vulnerable source code. Table 7 shows that 55 of the 346 analyzed dependent projects actually use the API from the vulnerable project. This highlights that there are still many systems (15.9%) that rely on libraries with known security vulnerabilities. Moreover, 18 of these 55 dependent projects not only include the API but also actually call the class which contains the vulnerable code. 2 out of these 18 dependent projects called and executed the vulnerable methods within the vulnerable projects.

```
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 25: Query to retrieve vulnerable code fragments across project boundaries

Table 7: Case Study #3 Results

Vulnerablity	Project	# Crawled Dependencies	# Actual usage	# Vuln. Classes usage	# Vuln. Methods usage
CVE-2015-0227	Apache WSS4J 1.6.16	242	15	10	0
CVE-2014-0050	Commons Fileupload 1.1	2	2	1	1
CVE-2014-0050	Commons Fileupload 1.2	102	38	7	1

6 Related Work

Given the diversity in technologies and software development processes, software artifacts often end up disconnected from each other, making it difficult for programmers to locate knowledge relevant to their specific development task. While the MSR community has made significant progress in analyzing individual repositories, the MSR community has yet to address the issue of seamless integrating these knowledge resources [30]. 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 software engineering, ontologies have been used to support requirement management [41], traceability [42], and use case management [43]. In the software testing domain, KITSS [44] 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. [45] 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. In [46] the KOntoR system was introduced to store semantic descriptions of components in a knowledge base and supports the semantic querying of this knowledge. In [47], Jin et al. discuss an ontological approach of service sharing among program comprehension tools. Hyland-Wood et al. [48] proposed an OWL ontology of software engineering concepts, including classes, tests, metrics, and requirements. Bertoa et al. [49] focused on software measurement. Witte et al. [50] used text mining and static code analysis to map documentation to source code in RDF for software maintenance purposes. Yu et al. [51] represented static source code information using an ontology and SWRL rules to identify common bugs in source code. In [54], Olszewska et al. introduce a general ontological modeling approach to provide a general support and guide the selection of software lifecycle artifacts during the software development life cycle.

Several researchers have described software evolution artifacts extracted from existing software repositories as OWL ontologies to facilitate everyday repository mining activities. Schlutter et al [58] present a general knowledge extraction approach based on an explicit knowledge representation of the content of natural language requirements as a semantic relation graph. Their approach is fully automated and includes an NLP pipeline to transform unrestricted natural language requirements into a graph. Kiefer et al. presented EvoOnt [52], 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 [53] to provide RDF-based access to JIRA bug trackers, Subversion, developer blogs, and project mailing lists. Wursch et al. presented SEON [30], 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. In [59] Zhou et al. introduces DockerKG, a tool to constructing a knowledge graph of Docker artifacts, which includes sources of software packages, their relationships and information of package installation procedures and operating systems. While the approach by Zhou et al [59] introduces a knowledge graph for Docker artifacts, their objective differs from ours in terms of providing a more generic dependency management and software analytics approach, which provides an extensible knowledge modeling approach, which focuses on the use of SW inference services and allowing for the full support of the FAIR knowledge and data modeling principle.

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. Given this standardized knowledge representation, we can envision interesting interactions among our semantics-aware analysis, ontologies and knowledge graphs introduced by others (e.g., [59]). Such extensions could lead to an entirely new family of software analysis services or at least simplify and enhance the implementation of existing ones.

7 Threats to Validity

Quality of our Ontology Design. One major benefit of our approach is its ability to integrate and reuse ontologies. However, assessing the quality of 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! and the Neon Toolkit) to check our ontology design for taxonomic, syntactical and consistency problems. To determine if our ontology constraints are 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 due to missing license information and annotations (such as <rdfs:label> and <rdfs:comment>) for some of our ontology elements. As part of our ontology maintenance, we fixed these issues.

Another potential threat to our approach is whether the set of concepts we considered are sufficient to capture the semantics of the analyzed domain. There is always a trade-off in terms of expressivity and usefulness in the design of knowledge bases; an equilibrium should be established between the amount of information needed to accomplish a task and the granularity of the knowledge that should be available to produce useful results. We partly addressed this threat through our case studies, which illustrate that our modeled build and dependency management concepts are sufficient to support different types of dependency analysis.

Generalizability. The case studies described in this research 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. Our domain specific ontologies are generic in nature and can be extended to specific system level ontologies. For example, our domain level ontologies for build management systems captures only core concepts and dependencies found commonly in the domain of build management systems. As part of our modeling approach, these core concepts can be further extended and enriched at the system level, without having to change our overall ontology design. For this research, we do model the domain of object-oriented programming languages, software vulnerabilities, software licenses, and build repositories as individual domains of discourse and provide concrete system level extensions for Maven, NVD and Java.

8 Conclusion

The software engineering landscape has changed over the last decade with projects and organizations increasingly taking advantage of the plethora of features and functionality provided by existing third-party libraries and components. Despite the existing role of build and dependency management systems, little is known on how this software dependency information can be integrated with other software-related knowledge to improve software development processes. In this research, we argue 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 address this, we present 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. It 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 improve knowledge integration and reuse. We further discuss the integration of additional knowledge sources with SBSON and illustrate the applicability of our approach in analyzing the impact of code reuse from a dependency management perspective. As part of our future work, we plan to integrate crowd-based knowledge sources (e.g., blogs, online video tutorials, Q/A forums) with our model to derive new applications.

References

[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] 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.

[10] 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.

[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] 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.

[28] I. Sonatype, *Maven: The Definitive Guide*. O'Reilly, 2008.

[29] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Sci. Am.*, vol. 284, no. 5, pp. 34–43, May 2001.

[30] 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.

[31] 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.
[32] 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.
[33] N. Noy and D. McGuinness, “Ontology Development 101: A Guide to Creating Your First Ontology,” 2001.
[34] 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.
[35] M. Uschold and M. Gruninger, “Ontologies: principles, methods and applications,” *Knowl. Eng. Rev.*, vol. 11, no. 02, p. 93, Jun. 1996.
[36] S. S. Alqahtani, E. E. Eghan, and J. Rilling, “SE-GPS,” 2015. [Online]. Available: http://aseg.encs.concordia.ca/segps/. [Accessed: 05-Jan-2019].
[37] J. Williams and A. Dabirsiaghi, “The unfortunate reality of insecure libraries,” *Asp. Secur. Inc*, pp. 1–26, 2012.
[38] 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.
[39] 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.
[40] NIST, “National Vulnerability Database,” 2007. .
[41] 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.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] 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.
[47] 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.
[48] 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.
[49] 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.
[50] R. Witte, Y. Zhang, and J. Rilling, “Empowering software maintainers with semantic web technologies,” *Eur. Conf. Semant. Web Res. Appl.*, pp. 37–52, 2007.
[51] 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.
[52] 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.
[53] 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.
[54] J. Olszewska and I. Allison (2018), "ODYSSEY: Software Development Life Cycle Ontology", in *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - Volume 2: KEOD,* ISBN 978-989-758-330-8, pages 303-311. DOI: 10.5220/0006957703030311
[55] Y. Zhou, X. Yang, T. Chen, Z. Huang, X. Ma and H. C. Gall, "Boosting API Recommendation with Implicit Feedback," in *IEEE Transactions on Software Engineering*, doi: 10.1109/TSE.2021.3053111.
[56] K. Thayer, S. E. Chasins, and A. J. Ko. 2021. A Theory of Robust API Knowledge. *ACM Trans. Comput. Educ.* 21, 1, Article 8 (March 2021), 32 pages. DOI:https://doi.org/10.1145/3444945
[57] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang and M. Zhou, "A Multi-Metric Ranking Approach for Library Migration Recommendations," *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 72-83, doi: 10.1109/SANER50967.2021.00016.
[58] A. Schlutter and A. Vogelsang. 2020. Knowledge Extraction from Natural Language Requirements into a Semantic Relation Graph. In *Proceedings of the IEEE/ACM 42nd Int. Conf. on Software Engineering Workshops* (*ICSEW'20*). Association for Computing Machinery, New York, NY, USA, 373–379. DOI:https://doi.org/10.1145/3387940.3392162
[59] Jiahong Zhou, Wei Chen, Chang Liu, Jiaxin Zhu, Guoquan Wu, and Jun Wei. 2020. DockerKG: A Knowledge Graph of Docker Artifacts. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (*ICSEW'20*). Association for Computing Machinery, New York, NY, USA, 367–372. DOI:https://doi.org/10.1145/3387940.3392161