Software Quality Journal
https://doi.org/10.1007/s11219-018-9428-4

API trustworthiness: an ontological approach for software library adoption

Ellis E. Eghan[1] · Sultan S. Alqahtani[1] · Christopher Forbes[1] · Juergen Rilling[1]

Published online: 04 February 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
The globalization of the software industry has led to an emerging trend where software systems depend increasingly on the use of external open-source external libraries and application programming interfaces (APIs). While a significant body of research exists on identifying and recommending potentially reusable libraries to end users, very little is known on the potential direct and indirect impact of these external library recommendations on the quality and trustworthiness of a client's project. In our research, we introduce a novel Ontological Trustworthiness Assessment Model (OntTAM), which supports (1) the automated analysis and assessment of quality attributes related to the trustworthiness of libraries and APIs in open-source systems and (2) provides developers with additional insights into the potential impact of reused libraries and APIs on the quality and trustworthiness of their project. We illustrate the applicability of our approach, by assessing the trustworthiness of libraries in terms of their API breaking changes, security vulnerabilities, and license violations and their potential impact on client projects.

Keywords Software quality · Trustworthiness · Code reuse · License violations · API breaking changes · Software security vulnerabilities

✉ Juergen Rilling
juergen.rilling@concordia.ca

Ellis E. Eghan
e_eghan@encs.concordia.ca

Sultan S. Alqahtani
s_alqaht@encs.concordia.ca

Christopher Forbes
c_forb@encs.concordia.ca

[1] Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada

1 Introduction

Traditional software development processes, with their focus on closed architectures, platform-dependent tools and software, restrict potential code reuse. With the introduction of the Internet, these restrictions have been removed, allowing for global access, online collaboration, information sharing, and internationalization of the software industry (Gao et al. 1999). Software development and maintenance tasks can now be shared among team members working across and outside organizational boundaries. Code reuse through resources such as software libraries, components, services, design patterns, and frameworks published on the Internet has become an essential aspect of this global code reuse and sharing among developers and organizations within the software engineering industry. Most of today's software projects increasingly depend on the usage of external libraries, which allows software developers to take advantage of features provided by application programming interfaces (APIs) without having to reinvent the wheel. Unfortunately, even though third-party libraries are readily available, developers are faced with new challenges with this new form of code reuse, such as being unaware of the existence of libraries, selecting the most relevant library among several possible alternatives, and how to use features provided by these libraries (Thung et al. 2013; Rahman et al. 2016).

Several software library recommendation approaches have been proposed to address these challenges. These approaches fall into two main categories: (1) recommendation systems for libraries and APIs based on characteristics such as popularity (Mileva et al. 2010), frequency of migration (Teyton et al. 2012; Hora and Valente 2015), and stability (Raemaekers et al. 2012), without considering the context of use of these libraries, and (2) techniques that take a client's context into account when recommending libraries (e.g., using the history of method usages by developers; McCarey et al. 2005).

However, reused software libraries should not only satisfy a client's functional requirements; they must also satisfy non-functional requirements (NFRs) such as security, safety, and dependability (Parnas 1994), which are critical to the success of software systems. NFRs are often referred to as system qualities and can be divided into two main categories: (1) execution qualities—qualities which are observable at runtime (e.g., performance and usability), and (2) evolution qualities, such as testability, trustworthiness, maintainability, extensibility, and scalability, which are embodied in the static structure of a software system. NFRs often play a critical role in the acceptance and trust users will have in a final software product. However, assessing and evaluating the trustworthiness of today's software systems and software ecosystems remains a challenge due to issues ranging from a lack of traceability among software artifacts to limited tool support.

Trustworthiness is also an inherently subjective and ubiquitous term since its interpretation depends on the assessment context of the stakeholder, which might be different among stakeholders and the context of use in which the library is used. Assessment models, therefore, should provide the flexibility and customizability to take into account such specific application contexts and the particular assessment needs of stakeholders (Hmood et al. 2012).

In our prior works (Alqahtani et al. 2017; Alqahtani et al. 2016), we introduced our Security Vulnerabilities Analysis Framework (SV-AF), a semantic modeling approach which establishes traceability links between security and software databases such as build repositories and version control repositories. We also introduced a generic quality assessment model (SE-EQUAM) (Hmood et al. 2012) which uses ontologies to model and conceptualize quality factors, sub-factors, attributes, measures, weights, and relationships used to assess software

quality. The work in this paper is a continuation of these previous works on semantic modeling and tracing of software security vulnerabilities, semantic analysis, and quality assessment. In what follows, we present our Ontology-Based Trustworthiness Assessment Model (OntTAM), which is an instantiation and extension of our SE-EQUAM assessment model (Hmood et al. 2012), for the domain of software library trustworthiness. OntTAM supports new semantic analysis for software license compatibility and the impact of API breaking changes, as well as our existing vulnerability analysis.

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

Our research is significant for several reasons:

1) We introduce OntTAM, a novel trustworthiness assessment model that takes advantage of both our previous generic SE-EQUAM software assessment model (Hmood et al. 2012) and our unified ontological knowledge representation of different SE-related knowledge resources (Hmood et al. 2012; Alqahtani et al. 2017; Alqahtani et al. 2016) while supporting the customization of the model to meet a stakeholder's assessment needs.
2) We introduce as part of OntTAM, novel trustworthiness measures, which measure API breaking changes, security vulnerabilities, and license violations. These measures take advantage of our ontologies and semantic reasoning services to allow for a trustworthiness analysis across the boundaries of individual artifacts and projects.
3) We report on a case study that illustrating how our approach can be applied to assess the trustworthiness of OSS libraries and discuss the potential impact of these libraries on the trustworthiness of the overall system.

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 the implications of reusing software libraries, the semantic web technologies, and an existing framework for analyzing software quality. We present our research overview and methodology in Section 4. Case study experiments and a discussion of our findings are presented in Section 5. 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 Motivating example

In what follows, we introduce a motivating example (Fig. 1) describing how our fictional software developer (Bob) attempts to reuse external libraries while facing several challenges during selecting the best library for his project while reducing their negative effect on the trustworthiness of his own project.

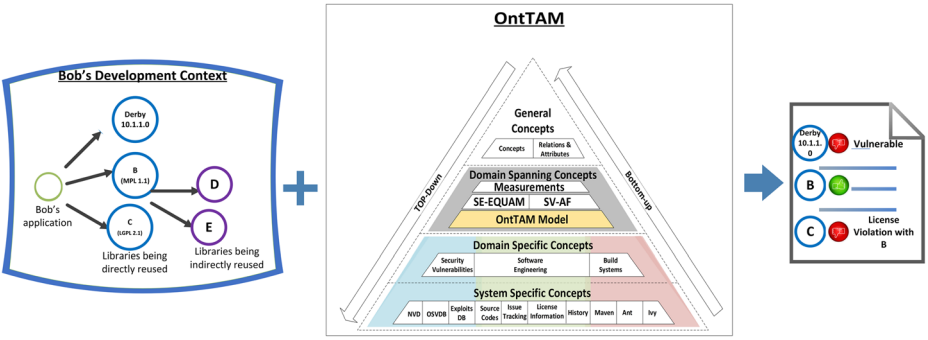


Fig. 1 Motivating example—how OntTAM can assist developers in trust assessment

Bob is currently developing an application which requires an embedded database. Bob tries to reduce his development effort, by searching the Internet for possible third-party libraries and components which meet his work context. His search returns Apache Derby,[1] an open-source-embedded database management systems (DBMS) implemented entirely in Java. However, Bob is faced now with the dilemma of deciding upon which version of Derby he should be using—the most recent (Derby version 10.11.1.1) or the most widely used one (Derby version 10.1.1.0). Following the recommendations published in the existing research (e.g., Mileva et al. 2009), Bob decides to use an older version of Apache Derby (version 10.1.1.0) due to its widespread usage/popularity. However, this recommendation results in the reuse of a component, which contains three known security vulnerabilities that are already reported in the National Vulnerability Database (NVD) (Table 1). In contrast, the newer version of Derby (version 10.11.1.1) does not contain any known vulnerabilities.

However, this is not the only risk Bob is susceptible to when selecting a library. Derby is licensed under the Apache 2 copyright license; for Bob not to introduce any license violation or incompatibility, he has to make sure that the selected library is compliant with his project license. For example, one cannot combine code released under the Apache 2 license with code released under the GNU GPL 2 (F. S. Foundation 2014).

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

3 Background

3.1 External library reuse and its implications on project quality

As previously discussed, reuse of functionality provided by third-party (external) software libraries is a growing trend in the software development industry. Automated dependency management features

[1] db.apache.org/derby/

Table 1 Example of Derby versions and their depedent projects in Maven

Derby version	Release year	Reported vulnerabilities in NVD	Direct dependencies in Maven repository
10.1.1.0	2005	3	382
10.5.3.0	2009	1	0
10.11.1.1	2014	0	36

have been introduced in modern build systems to simplify the integration and reuse of external libraries during development. Developers no longer have to manually manage their dependencies on software libraries. Build systems and dependency management tools automatically download and manage all required dependent components (including transitive dependencies) and perform any necessary dependency mediation (conflict resolution) when multiple versions of a dependency are encountered. Although this relieves developers from some of the dependency management, there remains an increased risk of including libraries which can negatively affect a project's overall quality and trustworthiness. In our research, we particularly consider the following quality risks introduced by software libraries: API breaking changes, security vulnerabilities, and license violations. In what follows, we briefly introduce background information about API breaking changes, security vulnerabilities, and license violations.

3.1.1 API breaking changes

Software libraries take advantage of visibility modifiers (e.g., public and protected modifiers in Java) to provide reusable and extendable APIs to other applications. However, these software libraries, as other software components, are subtle to change as they evolve over time. Unfortunately, the cost of evolving libraries may become higher, since such changes might impact many external clients. API changes can be classified into breaking and non-breaking changes (see Table 2) and can be defined as follows (Xavier et al. 2017):

- *Breaking changes*: any change that breaks backward compatibility through removal or modification of API elements, resulting in compilation errors in the client projects after the API update.
- *Non-breaking changes*: changes that preserve compatibility and usually involve the addition of new functionalities to the library. Thus, allowing to migrate between API versions which include only non-breaking changes does not cause negative effects to client applications.

Table 2 Breaking and non-breaking changes

Category	API element	List of changes
Breaking	Type	Removal, visibility loss, super-type change
	Field	Removal, visibility loss (e.g., public to private), type change (e.g., double to integer), default value change
	Method	Removal, visibility loss, return type change (e.g., Boolean to void), parameter list change, exception list change
Non-breaking	All	Addition, visibility gain (e.g., from private to public or protected), deprecation (e.g., deprecated method removal)

Table 3 The most common breaking and non-breaking changes in the Maven repository (Raemaekers et al. 2014)

Breaking changes			Non-breaking changes		
No.	Change type	Frequency	No.	Change type	Frequency
1	Method removed	177,480	1	Method added	518,690
2	Class removed	168,743	2	Class added	216,117
3	Field removed	126,334	3	Field added	206,851
4	Parameter type change	69,335	4	Interface added	32,569
5	Method return type change	54,742	5	Method removed, inherited still exists	25,170
6	Interface removed	46,852	6	Field accessibility increased	24,954
7	Number of arguments changed	42,286	7	Value of compile-time constant changed	16,768
8	Method added to interface	28,833	8	Method accessibility increased	14,630
9	Field type changed	27,306	9	Addition to list of superclasses	13,497
10	Constant field removed	12,979	10	Method no longer final	9202

Table 3 shows the top 10 breaking and non-breaking changes in the Maven repository as reported by Raemaekers et al. (2014). These breaking changes are obtained from 126,070 pairs of current and next versions of software libraries hosted in the Maven repository. The most frequent observed breaking change is method removals (177,480 observed occurrences). A method removal is considered to be a breaking change if the removal leads to compilation errors in places where this method is used. The most frequently non-breaking API change is method additions, with 518,690 occurrences. Although performing a change to a library might be a straightforward task, resulting breaking changes can have a significant ripple effect, which often will not only affect a single dependent class but even complete ecosystems.

3.1.2 Software security vulnerabilities

In the software security domain, a software vulnerability refers to mistakes or facts related to security problems in software, networks, computers, or servers. Such vulnerabilities represent security risks that can be exploited by hackers to gain access to system information or capabilities (Williams and Dabirsiaghi 2012). Among these systems, reuse of software libraries poses a significant threat, since vulnerabilities in a single component might affect many different systems across the globe.

Advisory databases (e.g., NVD) were introduced to provide a central place for reporting vulnerabilities and standardize their reporting and to raise developer awareness about the existence of such vulnerabilities. These databases rely on the Common Vulnerabilities and Exposures (CVE)[2] dataset, a publicly available dictionary for vulnerabilities which allow for a more consistent and concise use of security terminology in the software domain. Once a new vulnerability is revealed and verified by security experts, information about this vulnerability (e.g., unique identifier, source URL, vendor URL, affected resources, and related vulnerabilities information) are added to the CVE database. In addition to the CVE entry, each vulnerability will also be classified using the Common Weakness Enumeration (CWE)[3] database. The CWE provides a common language to describe and classify software security vulnerabilities based on their type of weakness. NVD, CVE, and CWE can be considered as being part of a global effort to manage the reporting and classification of known software vulnerabilities.

[2] https://cve.mitre.org/
[3] https://cwe.mitre.org/

While these databases are knowledge-rich resources, they often remain information silos, being disconnected from other knowledge in the software development domain (e.g., code repositories or issue trackers). These information silos are caused by (1) a lack of standardized formalism for representing knowledge in the software engineering domain, (2) the resulting inability to integrate seamlessly heterogeneous knowledge resources that would allow for both establishing semantic links across existing knowledge and inferring new knowledge, and (3) the lack of uniform resource identifiers that would support fact and analysis results sharing for consumption by either humans or machines across knowledge resource boundaries.

3.1.3 License violations

While dependency management tools, such as RubyGems,[4] Maven,[5] or CocoaPods,[6] have been introduced to automate the downloading and importing of libraries into projects, these libraries still originate from various authors and come with a plethora of OSS licenses (horizontal increase). One library can utilize another library, leading to hierarchies of libraries and license dependencies. All of these libraries' licenses must be compatible and compliant with each other. License violations and incompatibilities are an often overlooked factor when recommending licenses and therefore can significantly impact the trustworthiness of software systems. When incompatible licenses are used together, a license violation occurs. A license violation is defined as "the act of making use of a (licensed) work in a way that violates the rights expressed by the original creator" (Seneviratne et al. 2009). That is, not following the legal terms and conditions set out in the source license. Software authors who commit a license violation open themselves to the possibility of being sued; sometimes this risk can amount to millions of dollars.

3.2 Ontologies and Semantic Web

The Semantic Web has been defined by Berners-Lee et al. as "an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation" (Berners-Lee et al. 2001). It forms a Web from documents to data, where data should be accessed using the general Web architecture (e.g., URIs). Using this Semantic Web infrastructure allows data to be linked, just as documents (or portions of documents) are already, allowing data to be shared and reused across application, enterprise, and community boundaries. In a Semantic Web, data can be processed by computers as well as by humans, including inferring new relationships among pieces of data. For machines to understand and reason about knowledge, this knowledge needs to be represented in a well-defined, machine-readable language. Ontologies provide a formal and explicit way to specify concepts and relationships in a domain of discourse. The Semantic Web uses the Resource Description Framework (RDF) as its data model to formalize the meta-data as subject-predicate-object triples, which are stored in triplestores. Triplestores are database management systems (DBMS) for data modeled using RDF. Unlike relational database management systems (RDBMS), which store data in relations (or tables) and are queried using SQL, triplestores store RDF triples and are queried using SPARQL (Berners-Lee et al. 2001). The RDF data model is domain independent and users

[4] https://rubygems.org/
[5] search.maven.org
[6] https://cocoapods.org/

define ontologies using an ontology definition language. The Web Ontology Language (OWL) (McGuinness and Van Harmelen 2004) is an example of such a definition language and has been standardized by the W3C.[7] It supports the creation of machine-understandable information to enable Web resources to be automatically processed and integrated. The sub-language, OWL-DL, is based on description logics (DLs) (Mann 2003). DL is a logic-based formalism using predicate calculus to define facts that can formally describe a domain. Therefore, DLs are a set of axioms called a TBox (e.g., *Doctor* $\sqsubseteq$ *Person*) and set of facts called ABox (e.g., *{Parent(John), hasChild(John, Mary)}*). Both TBox and ABox form a knowledge base (KB) and often written $\mathcal{K} = \prec \mathcal{T}, \mathcal{A} \succ$. The RDF data model forms a graph where nodes (subject, object) are connected through edges (predicates). The SPARQL query language (DuCharme 2011) is used to retrieve information from RDF data model graphs.

Ontologies vs. models A model is "an abstraction that represents some view on reality, necessarily omitting details, and for a specific purpose" (Henderson-Sellers 2011). However, in SE, ontologies and models try to address the same problems (representing the software complexity in an abstract manner) but from very different perspectives. The differences between ontologies and models often result in different artifacts, uses, and possibilities. For example, modern SE practices advice developers to look for components that already exist when implementing functionality, since reuse can avoid rework, save money, and improve the overall system quality (Witte et al. 2007). In this example, ontologies can provide clear advantages over models in integrating information that normally resides isolated in several separate component descriptions. Furthermore, models (e.g., UML) rely on the closed world assumption, while ontologies (e.g., OWL) support open-world semantics. OWL, an example of ontology languages, is a "computational logic-based language" that supports full algorithmic decidability in its OWL-DL (description logic) variant. It is not possible to use algorithms supported by OWL (e.g., subsumption) for modeling languages due to their different semantics. Additional differences between ontologies and models are reported and discussed in Atkinson et al. (2006).

3.3 Evolvable quality assessment metamodel (SE-EQUAM)

Quality is a widely used term to evaluate the maturity of development processes within an organization. Defining quality allows organizations to specify and determine if a product has met certain non-functional and functional requirements. However, as Kitchenham (Hmood et al. 2010) states: "quality is hard to define, impossible to measure, easy to recognize." Unlike functional requirements, where a single analysis technique (e.g., use case modeling) is sufficient to identify essentially all requirements, the same analysis is not appropriate for all quality requirements. Quality, as defined by ISO 9000:2000 (Hmood et al. 2010), is the "degree to which a set of inherent characteristics fulfills requirements," where a requirement is a "need or expectation that is stated, generally implied or obligatory."

Assessing the evolvability of software systems has been addressed in existing research through the introduction of software quality models, e.g., McCall et al. (1977), ISO/IEC 9126,[8] and QUALOSS (Bergel et al. 2009). These models share a common, while informal (not machine-readable), structural representation of software qualities (Fig. 2).

[7] https://www.w3.org/
[8] https://www.iso.org/obp/ui/#iso:std:39752:en

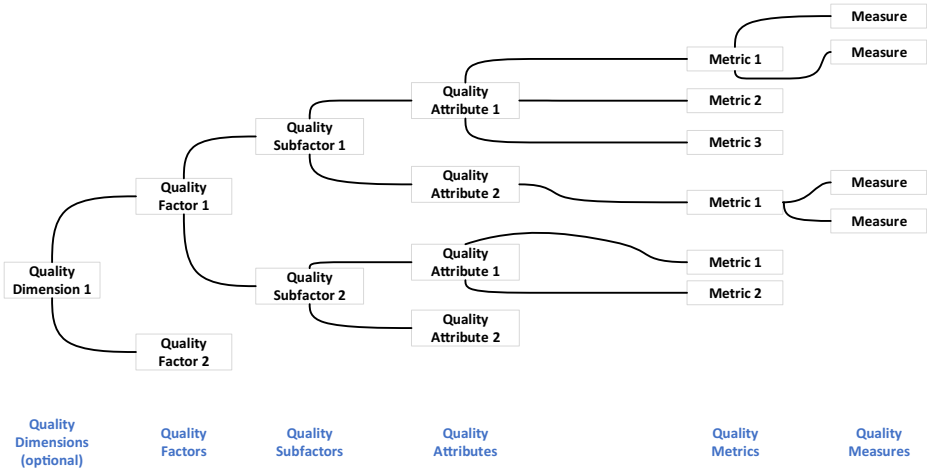


Fig. 2 Generic structure of quality assessment models (Hmood et al. 2010)

While these models are capable of assessing qualities in a given context, they lack the required formalism and semantics to allow them to evolve to meet the modeling requirements of different assessment contexts. The ability to adjust to change assessment needs was the main motivation for SE-EQUAM, an Evolvable QUAlity Meta-model that derives a formal (machine-readable) domain model that can adapt to changes in the assessment needs in terms of both artifacts being assessed and their assessment criteria (Hmood et al. 2012). SE-EQUAM addresses these challenges by taking advantage of the Semantic Web and its supporting technologies. SE-EQAM uses ontologies to model and conceptualize quality factors, subfactors, attributes, measures, weights, and relationships used to assess software quality. Input artifacts for the assessment model are various software artifacts such as version control systems and issue trackers, and its outputs are quality assessment scores based on the different assessment criteria. Ontologies not only provide a formal way to represent knowledge but also can eliminate ambiguity, enable validation, and provide a consistency-checking approach (Seedorf and Mannheim 2006). SE-EQUAM uses semantic reasoners to infer hidden relationships between domain model attributes. Given its formal representation, SE-EQUAM allows for its reuse by simplifying the instantiation of new domain-specific instances of the model. More details about the semantic reasoning are provided in Hmood et al. (2012).

Figure 3 illustrates the reuse and instantiation of our SE-EQAM model. The generic *syntactic meta-model*, which is a generic model that forms the basis for all quality models, can be instantiated by a domain model (e.g., ISO/IEC 9126). Furthermore, SE-EQUAM allows for a semantic mapping between the syntactical meta-model and a *semantic ontology meta-model*, which can then be instantiated as domain *model ontology* based on user-defined assessment criteria.

The SE-EQUAM process The general SE-EQUAM process (Fig. 4) represents a set of tasks and activities which we followed to allow for deriving a generic quality assessment method that can be used to customize and instantiate the generic model to meet a stakeholder's specific quality assessment context.

The input to the SE-EQUAM process is software artifacts and a set of core quality measurements applicable to these artifacts. In the next step, a common ontological representation for these artifacts has been established by reusing existing models or customizing existing

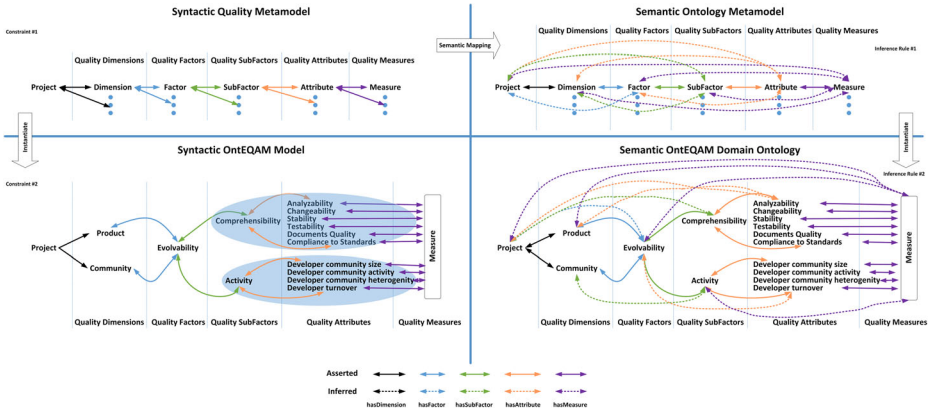


Fig. 3 SE-EQUAM ontology meta-model reuse to instantiate a domain model ontology (OntEQAM) (Hmood et al. 2012)

models to meet the requirements of these artifacts. As part of the model adjustment activity, quality metrics and measurements included in the core model can be customized and extended to reflect a specific model context. The output of this process is an instantiated assessment model, which meets specific user and project assessment requirements, by providing a quality assessment at both individual artifact and overall product level. Figure 4 illustrates the high-level activities and major tasks involved in the SE-EQUAM instantiation method.

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

4 Ontology-based trustworthiness assessment model (OntTAM)

OntTAM, an instantiation of the SE-EQUAM (Hmood et al. 2012) ontology meta-model, illustrates how our modeling approach can take advantage of the unified ontological representation of both software artifacts and the generic SE-EQUAM quality assessment model. OntTAM instantiates a domain-specific quality model to assess the trustworthiness of software projects and, more specifically, the trustworthiness of external libraries. OntTAM reuses SE-EQUAM's

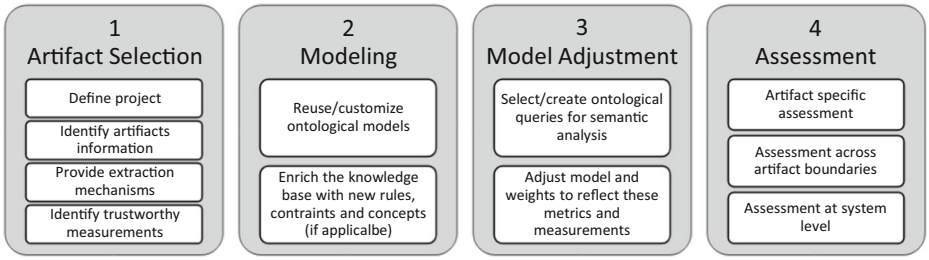


Fig. 4 SE-EQUAM process to instantiate evolvability model

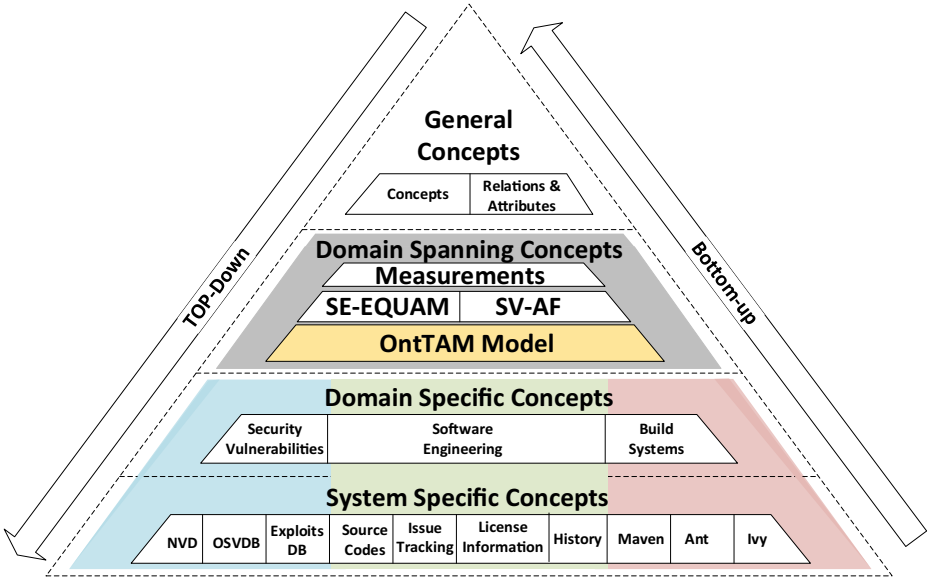


Fig. 5 The software security and trustworthy ontology hierarchy

core quality model structure which is based on quality factors, sub-factors, attributes, measures, weights, and relationships and extends them with trustworthiness-specific aspects. Inputs to OntTAM are knowledge resources such as version control systems, build systems, project license information, and security vulnerability information. The output of OntTAM is a trustworthiness assessment score for either an individual metric or an aggregation of sub-factors and factors for the overall product/library quality. The model thereby takes advantage of the OWL[9] and RDF/RDFS[10] semantic reasoning capabilities to infer hidden relationships between domain model attributes and to ensure the consistency among these attributes.

Figure 5 provides an overview of the knowledge model framework and its organization in terms of ontologies and their abstraction levels. While these ontologies may be derived modeled and used independently, a key objective of our approach is the knowledge integration across ontology boundaries, using both ontology alignments and semantic linking to create a unified ontological knowledge representation.

In what follows, we present our OntTAM methodology to further demonstrate how we instantiate different trustworthiness sub-factors (i.e., security, reliability, and legality), to establish a trustworthiness assessment for OSS products (e.g., external libraries). More specifically, we discuss in detail the four major steps involved in instantiating our customized OntTAM trustworthiness assessment model (Fig. 4): artifact selection, modeling, model adjustment, and the assessment process.

4.1 Artifact selection

The inputs to OntTAM are artifacts relevant to the reuse of software libraries within projects. These software artifacts can be categorized into endogenous and exogenous data.

[9] https://www.w3.org/OWL/
[10] https://www.w3.org/TR/rdf-schema/

Endogenous data represents data available internally to a software development environment (e.g., software artifacts related to versioning systems, issue trackers, software licenses, and build systems). Exogenous data refers in our context to data available externally to the software development environment (e.g., external vulnerabilities databases). Extracting and populating facts from these artifacts are often based on techniques commonly used by the MSR community (Kagdi et al. 2006; Kagdi et al. 2007; Kamiya et al. 2002). It should be noted that unstructured or semi-structured information (e.g., vulnerability descriptions and license information) often requires several preprocessing steps such as natural language analysis (NLP), as well as data cleansing to improve the quality of the data prior to the ontology population. More details about our data preprocessing and ontology population process can be found in Alqahtani et al. (2016) and Alqahtani et al. (2017).

4.2 Model and model adjustment

In this section, we discuss our knowledge modeling process in detail. It should be noted that in order to improve readability, we will be using the following prefixes as substitutes to the fully qualified names of our ontologies:

- rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
- owl: <http://www.w3.org/2002/07/owl#>
- seon: <http://se-on.org/ontologies/general/2012/02/main.owl#>
- sevont: <http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2015/02/vulnerabilities.owl#>
- sequam: <http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2017/09/sequam.owl#>
- onttam: <http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2017/09/onttam.owl#>
- sbson: <http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#>
- code: <http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/code.owl#>
- oswaldo: <http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2017/09/license.owl#>

4.2.1 Modeling project trustworthiness

Since OntTAM is based on the generic SE-EQUAM model, OntTAM is an extension and specialization of our core SE-EQUAM software quality assessment model. OntTAM is extended to provide a syntactical trustworthiness quality model that includes and defines a set of sub-factors, attributes, and metrics required for the assessment of trustworthiness. Many of these trustworthiness factors, attributes, and metrics are derived from existing work on trustworthiness assessment of open and closed source projects (Hmood et al. 2012; Hmood et al. 2010). The OntTAM-specific trustworthiness assessment is based on the two general quality dimensions, the community and product dimensions. The community dimension assesses the adoption of a software product by the community over an

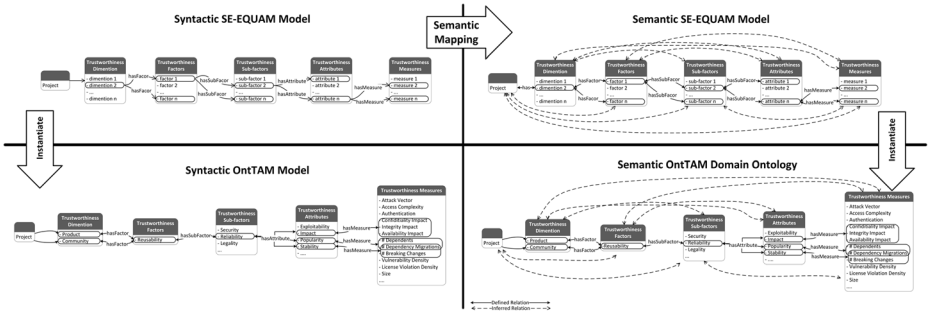


Fig. 6 Reuse of the SE-QUAM meta-model to instantiate the OntTAM domain model ontology

extended period of time, by considering the popularity in terms of downloads, rankings, and activity of the development community. The product dimension assesses the internal structure of the product and the development processes that impact its reusability which is the focus of this paper.

Figure 6 above provides an overview of the complete model instantiation process which provides as its output a formal (machine-readable) and semantic-enriched trustworthiness assessment model. The process involves applying both a syntactic and semantic mapping from SE-EQUAM to OntTAM. While the syntactical model allows us to answer basic queries such as: *What are the sub-factors associated with product trustworthiness?*, the semantic mapping enables the use of DL axioms (such as the property chain axiom) to infer new implicit relationships (dashed lines in Fig. 6 – semantic OntTAM ontology) from explicitly modeled relationships in OntTAM (solid lines in Fig. 6).

Figure 7 illustrates the main steps which are applied to associated trustworthiness concepts and measures for a sample project (ProjectX):

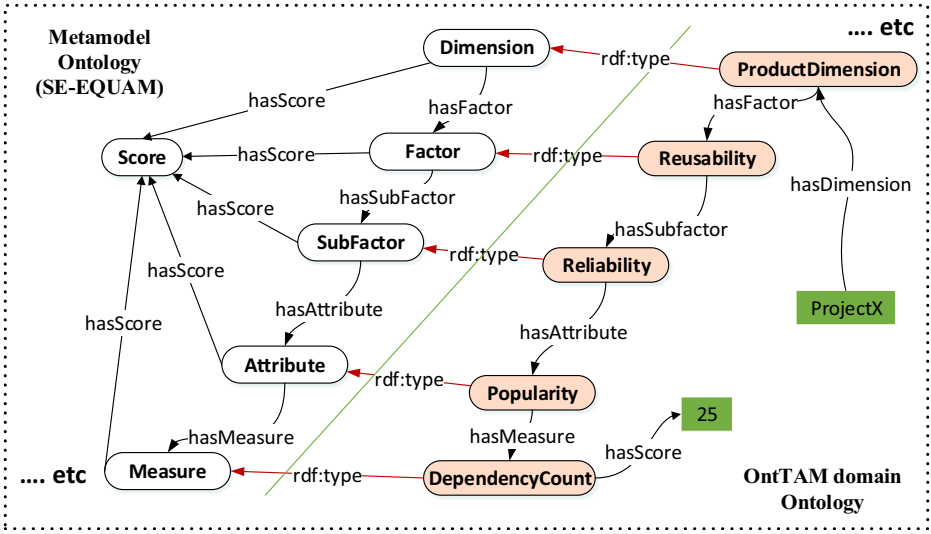


Fig. 7 An example defining the associated trustworthiness concepts and measures for a sample project

1. Define the *product* and *community* dimensions.

```
<onttam:ProductDimension><rdfs:type><sequam:Dimension> and
<onttam:CommunityDimension><rdfs:type><sequam:Dimension>.
```

2. Define *reusability* as a factor that is associated with the *product* dimension.

```
<onttam:ProductDimension><sequam:hasFactor><onttam:Reusability> and
<onttam:Reusability><rdfs:type><sequam:Factor>.
```

3. Following the same approach, OntTAM defines *reliability* as a sub-factor of *reusability* which is associated with the *popularity* attribute.

```
<onttam:Reusability><sequam:hasSubfactor><onttam:Reliability>,
<onttam:Reliability><rdfs:type><sequam:Subfacor>,
<onttam:Reliability><sequam:hasAttribute><onttam:Popularity> and
<onttam:Popularity><rdfs:type><sequam:Attribute>.
```

4. Assuming that OntTAM assesses a product's reusability through the *popularity* trustworthy attribute using the *DependencyCount* measure, we can now define this as:

```
<onttam:Popularity><seon:hasMeasure><sbson:DependencyCount>and
<sbson:DependencyCount><rdfs:type><seon:Measure>.
```

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

- Project-related OWL 2 property chain constructs:

SubPropertyOf(ObjectPropertyChain(:Project :hasFactor) :Factor)
SubPropertyOf(ObjectPropertyChain(:Project :hasSubfactor) :Subfactor)
SubPropertyOf(ObjectPropertyChain(:Project :hasAttribute) :Attribute)
SubPropertyOf(ObjectPropertyChain(:Project :hasMeasure) :Measure)

- Dimension-related OWL 2 property chain constructs:

SubPropertyOf(ObjectPropertyChain(:Dimension :hasSubfactor) :Subfactor)
SubPropertyOf(ObjectPropertyChain(:Dimension :hasAttribute) :Attribute)
SubPropertyOf(ObjectPropertyChain(:Dimension :hasMeasure) :Measure)

- Factor-related OWL 2 property chain constructs:

SubPropertyOf(ObjectPropertyChain(:Factor :hasAttribute) :Attribute)
SubPropertyOf(ObjectPropertyChain(:Factor :hasMeasure) :Measure)

- Subfactor-related OWL 2 property chain constructs:

SubPropertyOf(ObjectPropertyChain(:Subfactor :hasMeasure) :Measure)

4.2.2 Integration with other knowledge artifacts

Assessing the overall trustworthiness of a software library requires us not only to instantiate OntTAM but also to integrate it with other ontological software knowledge artifacts to be able to derive and integrate novel trustworthiness measures. For the integration, we take advantage of software artifact ontologies we have created and refined over the years (Alqahtani et al. 2017; Zhang et al. 2008; Keivanloo et al. 2011) and by reusing existing ontologies (Würsch et al. 2012) that model different software artifacts. Figure 8 provides an overview of the unified ontological representation of software artifacts which we integrate with OntTAM. These artifacts include, but are not limited to, (a) Software Evolution Ontologies (SEON) which model software engineering repositories such as source code, version control systems, issue tracker systems, and licenses; (b) the Build Systems ONtology (SBSON) which captures knowledge about build management systems (e.g., Maven); and (c) the Software sEcurity Vulnerability Ontologies (SEVONT) for modeling software security vulnerability information such as severities, impacts, vulnerabilities types, and patch information found in different security databases.

The integration of these heterogeneous knowledge resources allows us to introduce different trustworthiness measures related to the reuse of software libraries. More specifically, in this research, we introduce the following three trust criteria: API breaking changes, security vulnerabilities, and license violations. Figure 8 shows the core concepts and object properties, distributed across the different abstraction layers of our knowledge modeling framework (Fig. 5). It should be noted the omitted data properties to improve the readability of the figure.

Among the core concepts used from these ontologies is the `BuildRelease` from the SBSON build ontology, which is a subclass of the `Release` concept which allows captures the fact that a project can have several releases (including library releases). A Release has a `License` and defines its dependencies on other releases. Each release contains a set of `CodeEntity` elements such as `Field`, `Method`, and `Class`. A release can be affected

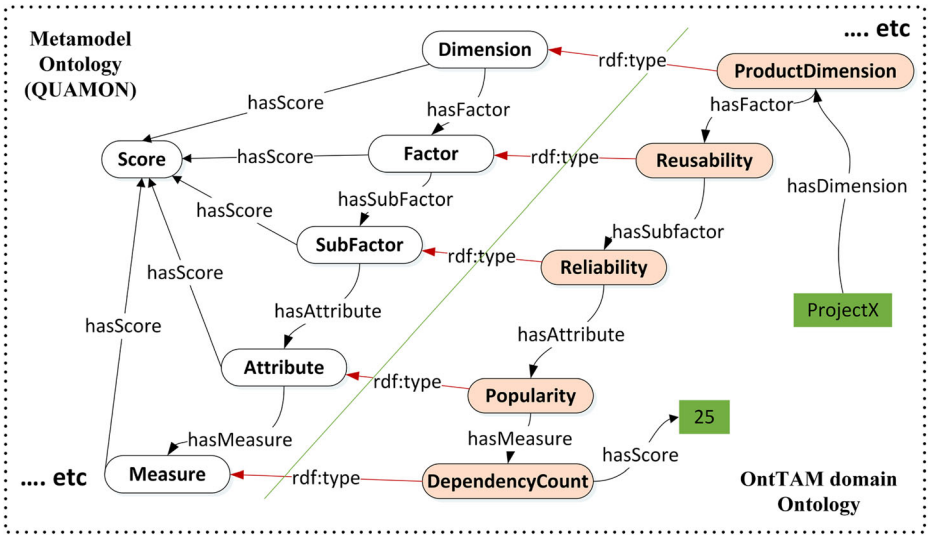


Fig. 8 Integrating OntTAM ontology into SV-AF model and reusing SE-QUAM concepts

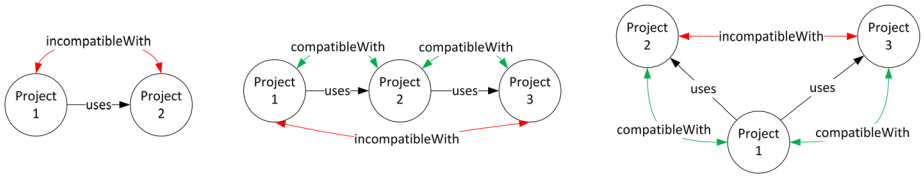
by a `Vulnerability`, leading to the release of a new version containing a `SecurityPatch`. A security patch corresponds to code changes introduced to fix some existing `VulnerableCode`, which is part of a `CodeEntity`. For example, if a class or method is modified during a security patch, then this code change can be used to locate the original `VulnerableCode`. The OWL classes, `SecurityPatch` and `VulnerableCode`, are linked in our model through an object property. For a complete description of the ontologies, how they are built, the alignment processes, and reasoning, we refer the reader to Alqahtani et al. (2016) and Alqahtani et al. (2017).

All of these core concepts have metrics used by the OntTAM assessment process. Measures have a *unit* and are expressed on a *scale*, e.g., an ordinal or nominal scale. Information about units and scales can be used to perform conversions (Würsch et al. 2012). Many base measures, such as the number of lines of vulnerable code (LOVC), number of known vulnerabilities, vulnerabilities severities (scores), and number of license violations provide, when viewed in isolation, only limited insights. Additional derived measures are needed to support further analysis and assessment of software artifacts. These derived measures represent an aggregation of values from different subdomains, for example, the number of vulnerabilities per class is an aggregation of measures derived from source code and the vulnerability repositories. While the abstract measurement concepts are defined in the general upper layer of our integrated model (Fig. 8), many *base measures* (e.g., size) and *derived measures* (e.g., weighted vulnerability density) are modeled in the domain-specific layer.

4.3 Measures and metrics

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

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



(a) Type 1 – Simple Violations (a) Type 2 – Transitive Violations (a) Type 3 – Compound Violations

Fig. 9 Categories of license violations

$$\mathrm{WVD}_{\mathrm{direct}}(\mathrm{release}) = \frac{\sum_{i=1}^{V} w_i}{S} \tag{1}$$

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

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

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

$$\mathrm{WVD}_{\mathrm{overall}}(\mathrm{release}) = \mathrm{WVD}_{\mathrm{direct}}(\mathrm{release}) + \mathrm{WVD}_{\mathrm{inherit}}(\mathrm{release}) \tag{3}$$

License violation count (*LVC*) is a measure to assess the number of license violations that exist within a given project. This measure can indicate potential long-term risks associated with intellectual rights violations that exist within a project. A license violation occurs if any of the dependent components of a parent project includes components with non-compatible licenses. Open source code license violations are often due to the fact that many software developers are simply neither aware nor well-versed in open source license compliance. For example, in 2008, the Free Software Foundation (FSF) claimed that various products sold by Cisco under the Linksys brand had violated the licensing terms of many programs on which FSF held the copyright.[11] These FSF programs were under the GNU General Public License, a copyleft license which allows users to modify a piece of software as long as the derivative work is under the same license.

In this work, we identify three (3) main categories of license violations: *simple violations*, *transitive violations*, and *compound violations* (see Fig. 9). $\mathrm{LVC}_{\mathrm{simple}}$, $\mathrm{LVC}_{\mathrm{transitive}}$, and $\mathrm{LVC}_{\mathrm{compound}}$ are base measures associated with each category. Details on how license violations are identified are presented in Section 5.3.

$$\mathrm{LVC}_{\mathrm{overall}}(\mathrm{release}) = \mathrm{LVC}_{\mathrm{simple}}(\mathrm{release}) + \mathrm{LVC}_{\mathrm{transitive}}(\mathrm{release}) + \mathrm{LVC}_{\mathrm{compound}} \tag{4}$$

Breaking change density (*BCD*) *metric* is a normalized measure which represents the ratio between breaking and non-breaking API changes that are introduced in a project. API changes often occur as a project and its components evolve inconsistently, resulting in incompatibilities of APIs and API calls. This measure can be used to determine the stability of an API over time—how often do breaking changes occur.

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

Details on how we identify breaking changes are presented in Section 5.4. The BCD metric can be represented formally as follows:

$$\mathrm{BCD} = \frac{\#\text{breaking API changes}}{\#\text{nonbreaking API Changes}} \tag{5}$$

Breaking change impact (*BCI*) measures the impact of breaking changes on client applications, by assessing a client application and its use of APIs with a changed contract. The impact of breaking changes on clients can be both direct and indirect. While there exists a significant body of work on the direct impact of changes (Raemaekers et al. 2012; Raemaekers et al. 2014; Robbes et al. 2012; Cossette and Walker 2012; Kapur et al. 2010), very little research has been conducted on indirect breaking changes. Indirect breaking changes occur, for example, when different versions of the same API are introduced by any of the client's other dependencies. By default, the Java virtual machine is unable to differentiate between multiple versions of the same API. In cases where multiple versions of a dependency are encountered, the first occurrence of an API version in a project's classpath is chosen. We introduce two measures that capture both direct and indirect breaking changes.

We represent the BCI metrics formally as follows:

$$\mathrm{BCI}_{\mathrm{direct}}(C, D) = \frac{\#\text{breaking API changes in } D \text{ used by } C}{\#\text{breaking API changes in } D} \tag{6}$$

$$\mathrm{BCI}_{\mathrm{indirect}}(C, < D_1, \ldots, \ D_n >) = \frac{\#\text{breaking API changes in } < D_1, \ldots, \ D_n > \text{ used by } C}{\#\text{breaking API Changes across } < D_1, \ldots, \ D_n >} \tag{7}$$

where C is the client project, D is the reused library, and $<D_1, \ldots, D_n>$ is the set of (direct and transitive) different library releases being reused by the client.

4.4 Assessment process

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

Figure 10 shows the set of transformation steps, which are performed during the fuzzification of the assessment process, with details of each step discussed in more details throughout the section.

1) *Measure calculation*: Input to this step are raw values from the populated ontologies. Measures are calculated by querying our populated knowledge base for the base and derived measures introduced in the previous section (e.g., *WVD*).
2) *Fuzzification*: The extracted quality measures and weight values are used to create fuzzy scales in the fuzzification step. As part of the fuzzification step, fuzzy scales are created for the different measures, the assessment weights (provided by stakeholders of the assessment model to assign a level of importance to different measures), and the overall assessment result. These results are converted to linguistic variables, which are variables whose values are expressed as words or sentences (values like high, not very high, low) (Zadeh 1975). These linguistic variables are the building blocks of fuzzy logic and become the input for the fuzzification inference engine.

Figure 11 shows an example of a fuzzy scale created for the WVD measure and its assessment weights. The *x*-axis represents the measurement results' range and the *y*-axis the membership degree (range is 0–1). The higher the membership value, the stronger the measurement's relation to its fuzzy result scales. The overlap between boundaries of categories in the fuzzy scale demonstrates the uncertainty in interpreting boundary measurement results.

Since high WVD, LVC, and BCD measures lower the overall quality and trustworthiness score of a project, we made the following three assumptions to automate the fuzzy inference rules for these measures: (1) in cases when the user-specified weight is high then the individual measure

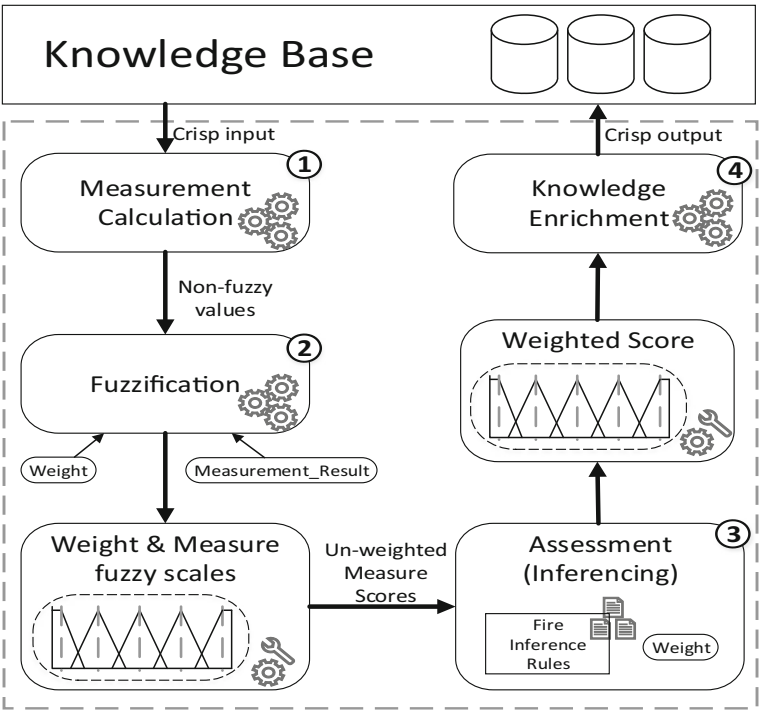


Fig. 10 Fuzzy assessment process steps

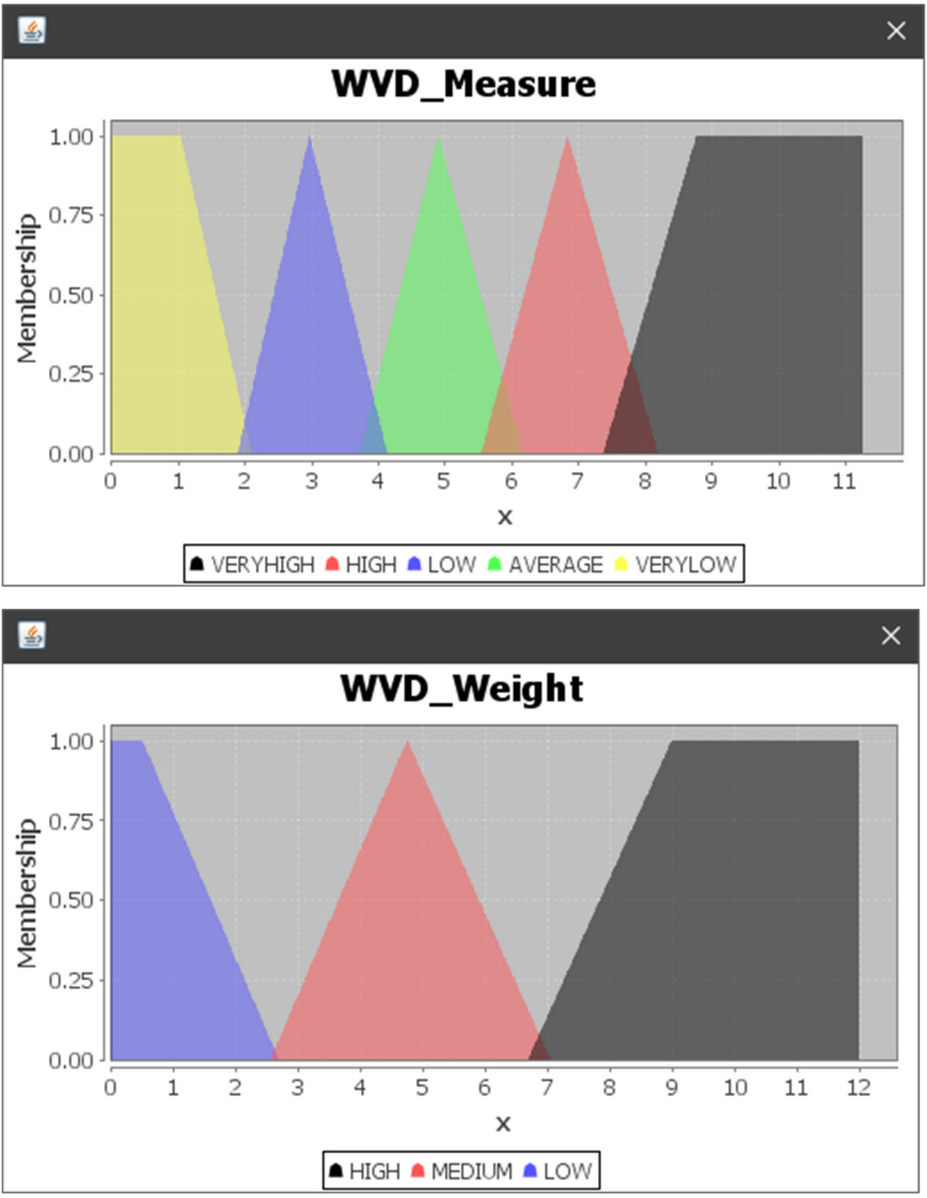


Fig. 11 WVD measure fuzzy scale and weight fuzzy scale for WVD measure

score is one level lower, *VeryPoor* scores will keep their values (e.g., a high weight will change an *Excellent* score to *VeryGood*); (2) the opposite holds for low weights, which reflects that their scores are less relevant to the overall assessment their scores are adjusted by one level higher. *Excellent* scores keep their values; (3) with medium weight, scores keep their values. These assumptions reflect the fact that when a measure is of high importance to the assessment (high weight), its score should be more sensitive to a low measure value.

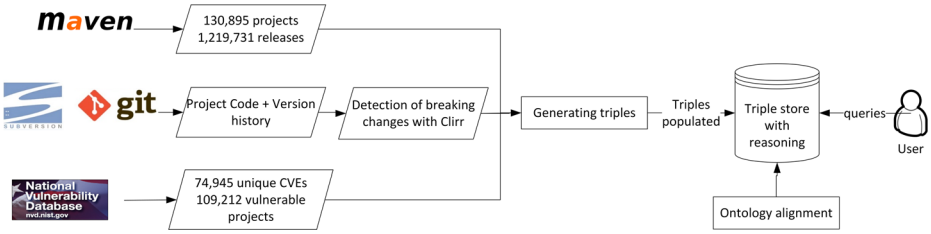


Fig. 12 Overview of case study setup process

3) *Inference and assessment:* Input for this step is the fuzzified measure and weight values in the form of linguistic variables. These linguistic results are now transformed into the final assessment score by executing a set of fuzzy inference rules. The de-fuzzification is based on a set of fuzzy inference rules, which are expressed in the Fuzzy Control Language (FCL) (I. E. Commission 2000) using the JFuzzyLogic inferencing engine (Cingolani and Alcala-Fdez 2012). The inference engine fires the relevant fuzzy rules based on the provided input. Firing rules will calculate the final weighted overall measurement result which is a combination of all the different measures. Using the center of gravity (COG) method, considered as one of the most popular de-fuzzification methods (Samoladas et al. 2008), the overall fuzzy measurement result is de-fuzzified back into a numerical assessment measurement results in order to be populated back to the knowledge base. As part of our assessment, we create a FCL file for each measure. The complete set of FCL files for all measures can be found online.[12]
4) *Knowledge enrichment*: This optional step allows for the integration of the assessment results at both the individual attribute, sub-factor, and overall assessment level. Our ontological representation enables us to seamlessly integrate these assessment results in the knowledge base, therefore not only supporting reuse of analysis results but also allowing their use for further semantic analysis.

5 Case study

The objective of this section is to demonstrate the applicability of our modeling approach to support the assessment of trust within OSS software libraries, by highlighting the flexibility of our modeling approach, in terms of its seamless knowledge and analysis results integration, as well as the use of Semantic Web reasoning services to infer new knowledge (measures). In Section 5.1, we present the setup for our study, including the selection process for the four projects used to illustrate our approach; Sections 5.2 to 5.4 describe how we identify and measure security vulnerabilities, license violations, and API breaking changes. Section 5.5 describes how these individual identified measures can be integrated for a holistic trustworthiness assessment.

5.1 Study setup

For the data collection and extraction in our case study (see Fig. 12), we rely on four data sources: the NVD database, GitHub, SVN, and the Maven build repository.

For our study, we have downloaded the latest versions of the Maven and NVD repositories—which include 1,219,731 project releases in Maven and 74,945 vulnerabilities

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

Table 4 Overview of selected case study projects

Project	No. of releases analyzed	No. of dependencies
Commons Fileupload	6	68,854
Apache CXF WS Security	5	4570
Struts	3	3170
ASM	20	8109

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

Fig. 13 The rules to infer the direct WVD measure

affecting 109,212 releases in NVD. For our study, we limited the assessment scope to four projects. The projects were selected based on the following criteria: (a) at least some of their releases contained known vulnerabilities, (b) license details were provided, (c) releases varied in their major version numbers, and (d) the functionalities these products provide are widely reused by other projects (see Table 4 for details). The four subject systems vary in size (classes and methods) and application domain. Commons Fileupload[13] adds file upload capabilities to web applications, and CXF WS Security[14] provides reusable components for client-side authentication, security, and encryption. Struts[15] is an open source framework for creating Java web applications, and ASM[16] is a Java bytecode manipulation library. We further extract the complete source code and history information of these four projects. The extracts facts were used to populate the corresponding ontologies and made persistent in our triplestore.

5.2 Identifying and measuring software security vulnerabilities

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

WVD_{direct} inference In order to derive the WVD_{direct} score for the projects, we define rules using the Semantic Web Rule Language (SWRL), similar to the one shown in Fig. 13. The rule states that, if some project release has a LOC and OverallSeverityScore measure, then the release has a WVD_{direct} score (obtained by dividing the overall severity score by LOC).

$WVD_{inherit}$ inference For us to be able to infer the $WVD_{inherit}$ measure of a project release, we had first to determine the ratio of vulnerable APIs that are reused in a particular release. The OntTAM knowledge model not only captures the required information to derive this measure, but also includes all semantics to be able to take advantage of the Semantic Web reasoners to

[13] https://commons.apache.org/proper/commons-fileupload/
[14] http://cxf.apache.org/docs/ws-security.html
[15] https://struts.apache.org/
[16] http://asm.ow2.org/

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

Fig. 14 SPARQL query for inferring the total number of vulnerable code entities in a project

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

Fig. 15 The SPARQL query for inferring the vulnerable code entities used by different dependent projects

infer the measure value. More specifically, once the required ontologies (e.g., SEVONT, SEON, OntTAM) are populated, a SPARQL query can be created to retrieve the number of vulnerable API elements in a given release (see Fig. 14).

Using Fig. 15, we can also determine the number of such vulnerable API elements being reused in client applications. For a more detailed description, on how we detect vulnerable code elements, the reader is referred to our previous work (Alqahtani et al. 2017).

The SPARQL query (Fig. 16) exemplifies how we take advantage of analysis results from the inference rules in Fig. 15 to infer the final $WVD_{inherit}$ measure for a particular release of a component.

```
CONSTRUCT{?client sevont:hasInheritWVD ?inheritWVD }
WHERE{
{
  SELECT ?client count(?indirectWVD) as ?inheritWVD
  WHERE {
   ?link a build:DependencyLink.
   ?link build:hasDependencySource ?client.
   ?link build:hasDependencyTarget ?release.
   ?client sevont:hasReusedVulnerableCodeCount ?usedVulnerableCodeCount.
   ?release sevont:hasVulnerableCodeCount ?totalVulnerableCodeCount.
   ?release sevont:hasDirectWVD ?directWVD.
   BIND((?usedVulnerableCodeCount/?totalVulnerableCodeCount) AS ?vulnerableCodeRatio).
   BIND((?vulnerableCodeRatio * ?directWVD) AS ?indirectWVD).
  }
}}
```

Fig. 16 SPARQL query for inferring inherited WVD measures in clients' projects

Table 5 Vulnerability densities of selected projects

Project	No. of vulnerabilities	Aggregated vulnerability scores	Size (KLOC)	WVD
commons-fileupload 1.0	2	10.8	1.23	8.78
commons-fileupload 1.1	2	10.8	1.28	8.46
commons-fileupload 1.2	2	10.8	1.78	6.05
commons-fileupload 1.2.1	2	10.8	1.97	5.49
commons-fileupload 1.2.2	2	10.8	2.04	5.31
commons-fileupload 1.3	1	7.5	2.39	3.14
Apache CXF WS Security 2.4.1	4	23.6	18.92	1.25
Apache CXF WS Security 2.4.4	4	23.6	21.30	1.11
Apache CXF WS Security 2.4.6	5	27.9	23.10	1.21
Apache CXF WS Security 2.6.3	8	39.4	26.43	1.49
Apache CXF WS Security 2.7.0	10	49.4	26.43	1.87
Struts 1.2.4	5	30	24.04	1.25
Struts 1.2.8	8	49.6	24.61	2.02
Struts 1.2.9	4	25.7	24.76	1.04

Findings and discussion Table 5 provides the analysis results for our case study in terms of known vulnerabilities, size, and WVD scores for selected project releases. Using the WVD measure, we can now compare two releases of the same project in terms of their weighted vulnerability density. For example, based on the WVD measure, we can consider Struts 1.2.9 to be more trustworthy than earlier versions of Struts (e.g., versions 1.2.4 and 1.2.8, which have both higher WVD scores). However, the latest version is not always better than earlier versions as seen with the analyzed Apache CXF WS Security libraries. Version 2.7.0 of the CXF WS Security library has a worse WVD compared to its previous versions—two new vulnerabilities were introduced in version 2.7.0 in addition to the existing vulnerabilities inherited from prior versions.

Table 6 Clients who switched from a vulnerable API in later release. *n/a* not available

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

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

Fig. 17 SPARQL query for inferring the total number of breaking changes in a project

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

Fig. 18 SPARQL query for inferring the total number of breaking changes in a project

We further analyzed the WVD results, to see whether developers actually migrate their applications to library versions which are less vulnerable (e.g., a newer version of the same library with patched vulnerabilities). Table 6 provides an overview of the number of dependent applications which change their build dependency to a more trustworthy release (based on the lower WVD score). Our analysis results show 45.1% client applications which switched their library dependencies; out of these, 63.29% switched to a more trustworthy library release. Surprisingly, the remaining 36.71% switched to library releases which are either equal or less trustworthy (higher WVD score), even if more trustworthy library versions are available.

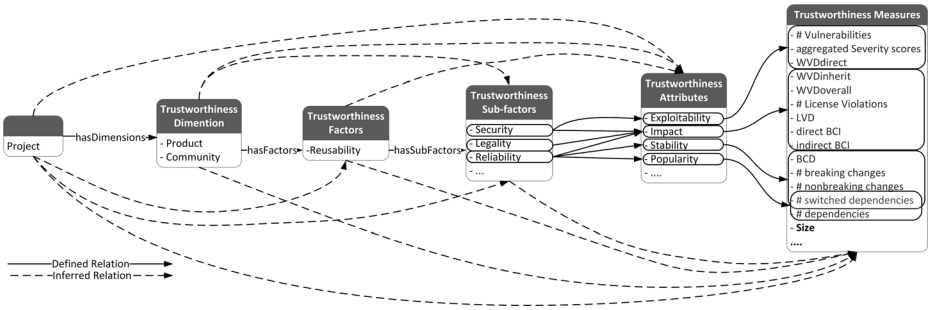


Fig. 19 SPARQL query for inferring the total number of breaking changes in a project

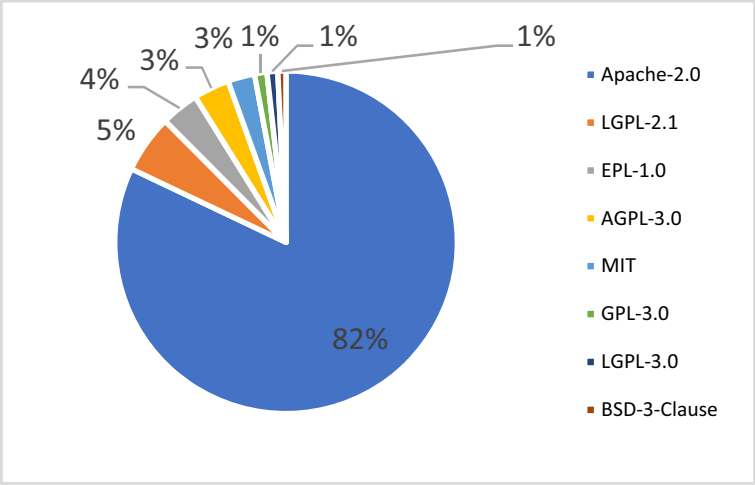


Fig. 20 License distribution in the Maven repository

5.3 Identifying and measuring license violations

Approach License violations originating from external libraries and components can cause a major long-term liability for client applications in terms of intellectual property and the trustworthiness of these libraries. In our study, we first evaluate if such license violations (non-compliances) occur in general in project dependencies managed by the Maven repository. In the second part of our study, we revisit our four projects used in our trustworthiness assessment study, to assess their trustworthiness in terms of license violations. For the study, we create SPARQL queries that analyze all dependency relationships in Maven and identify three (3) main categories of license violations: *simple violations*, *transitive violations*, and *compound violations* (see Section 4.3). The queries take advantage of both our open source license ontology and the build ontology. Figures 17, 18, and 19 illustrate the queries we used to identify these violations.

Findings and discussion This section presents and discusses the results obtained in our license violation experiment for the Maven repository. Figure 20 shows the distribution of common project licenses in the Maven repository, with Table 7 reporting on the license violations, classified by the type of violation, which we observed in our study of the Maven repository.

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

Table 7 Totals for each type of violation found by querying the data store

License violation types	Count
Type 1—simple violations	131,996
Type 2—embedded violations	288,153
Type 3—compound violations	654,964

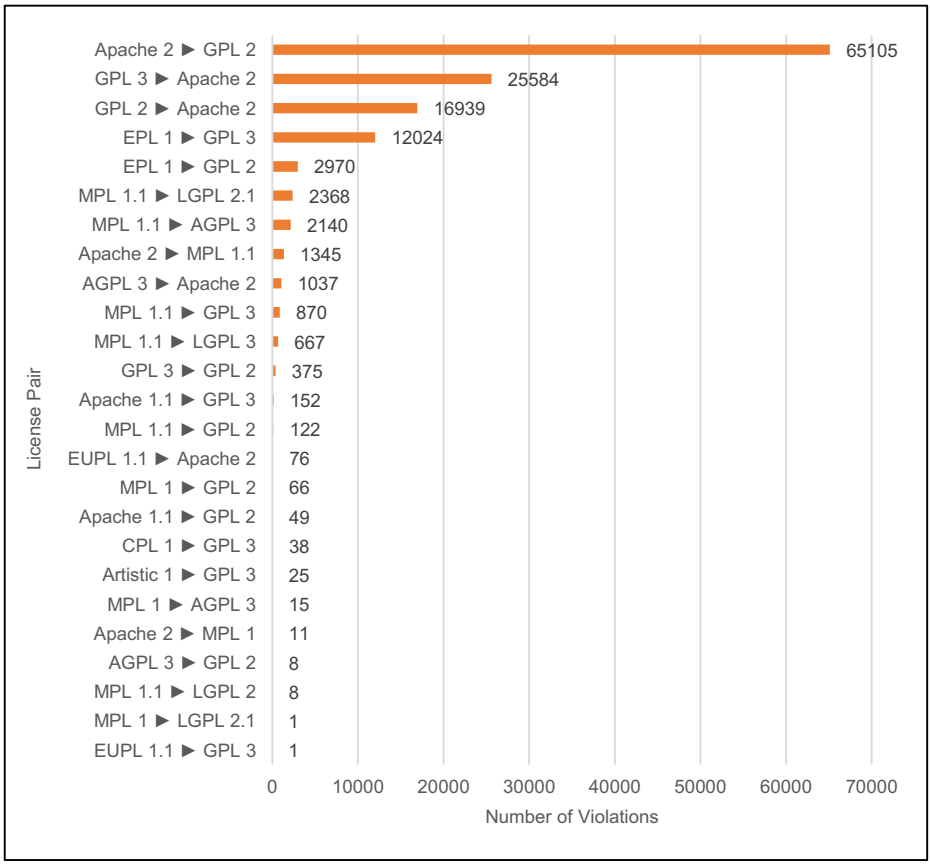


Fig. 21 Most popular Type 1 license violation pairs

Figures 21, 22, and 23 summarize the most common license violation pairs which occurred for all three license violation categories. The most common Type 1 violation which we observed is code published under the Apache 2 license being incorporated into GPL 2 licensed code. This violation is not surprising for two reasons. First, many software developers are simply not aware nor well-versed in open source license compliance, and as these are the two

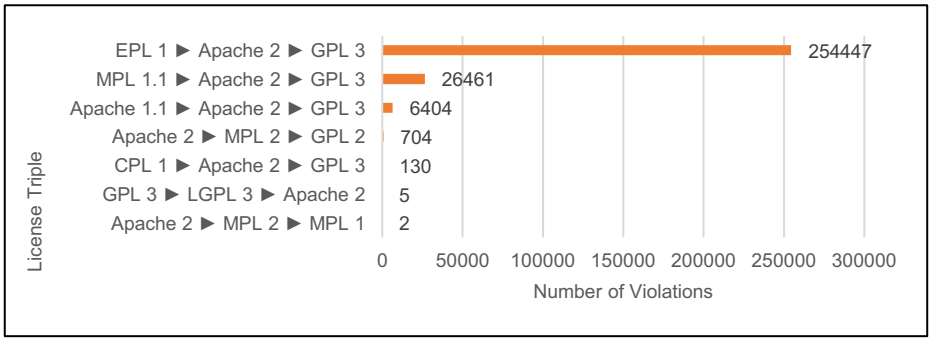


Fig. 22 Most popular Type 2 license violation pairs

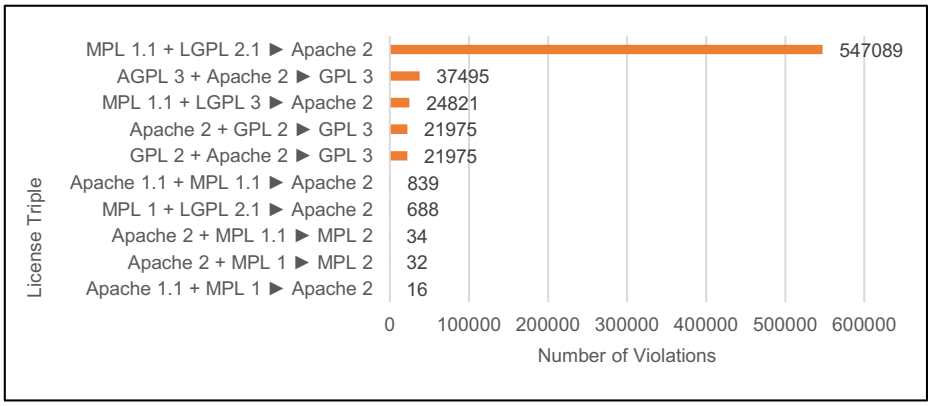


Fig. 23 Most popular Type 3 license violation pairs

of the most popular licenses in the world, this pairing reflects their usage in the wild. Second, there is likely some confusion about Apache 2's compatibility with the GPL. On the GNU website, the Free Software Foundation publishes a list of licenses that are compatible with the GPL. This page shows Apache 2 in green (meaning compatible), but in the license discussion, the authors explain that Apache 2 is only compatible with GPL 3, not GPL 2 (F. S. Foundation 2014).

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

Identification of Type 3 violations becomes even more difficult to detect since their detection largely depends on how licenses define derivative works and conditions for reusing these libraries. Libraries can be used by either including the actual source code or through linking (e.g., through a jar file). Linking of a library can be static (compile-time) or dynamic (run-time). For example, LGPL requires each project to be an "independent work that stands by itself, and includes no source code from [the other]." In this scenario, it is perfectly acceptable to combine the compiled code, however (Kuhn et al. 2016). So basically, the question is whether a derivative work is created or not, when combining dependencies into a new project. Derivative works come into play only when the licensed software is copied, distributed, or modified. Additional research is needed to further clarify legal and license compliance issue when using these open source licenses. However,

Table 8 License violation counts in selected projects

Project	No. of simple violations	No. of transitive violations	No. of compound violations
commons-fileupload 1.0	0	0	0
commons-fileupload 1.1	0	0	0
commons-fileupload 1.2	4	0	0
commons-fileupload 1.2.1	14	0	0
commons-fileupload 1.2.2	19	0	0
commons-fileupload 1.3	4	0	0
Apache CXF WS Security 2.4.1	0	0	0
Apache CXF WS Security 2.4.4	0	0	0
Apache CXF WS Security 2.4.6	0	0	0
Apache CXF WS Security 2.6.3	0	0	0
Apache CXF WS Security 2.7.0	0	0	0
Struts 1.2.4	0	0	0
Struts 1.2.8	0	0	0
Struts 1.2.9	0	0	0

as can be noted, all three types of violations can exist in projects. Thus, simple, transitive, and complex license violations are problems that occur in open source projects and can potentially affect the trustworthiness of components and libraries being reused in software projects.

In what follows, we report on license violation results which we observed for the selected four projects of our trustworthy study. Table 8 provides an overview of the number of license violations detected in these projects. Only four (4) releases of Commons Fileupload introduced violations in client applications. No license violations are reported for the projects due to the lack of license information in the analyzed client applications. Results, although incomplete, confirm our previous claim that violations are problems that occur in open source projects.

5.4 Identifying and measuring API breaking changes

Approach As previously mentioned in our study setup (Section 5.1, Fig. 12), we extract the source code and versioning information of the four projects from GitHub and SVN. For each successive pair of releases of a given project, we then identify the introduced breaking and non-breaking changes using the VTracker[17] tool. In order to be able to reuse the analysis results for further analysis, we take advantage of our ontological knowledge modeling approach and extend our knowledge base to include the analysis results. Developers can now access this information, using SPARQL queries, to derive potential direct and indirect impacts of breaking changes on their client applications. In what follows, we show some of the main rules and queries used to derive the BCD and BCI measures.

BCD inference For computing the BCD scores of the projects in our dataset, we define a SWRL rule (see Fig. 24), which infers the BCD score from the breaking and non-breaking change counts. Figures 25 and 26 detail the queries for computing the breaking and non-breaking change measures of a project.

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

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

Fig. 24 The rules to infer the BCD measure

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

Fig. 25 SPARQL query for inferring the total number of breaking changes in a project

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

Fig. 26 SPARQL query for inferring the total number of non-breaking changes in a project

BCI_{direct} and $BCI_{indirect}$ inference The queries in Figs. 27 and 28 take advantage of the inference services to derive both the direct and indirect BCI scores from a project and its dependencies. The query in Fig. 25 first identifies two unique releases of the same project for which breaking changes have been populated into the triplestore. It then identifies any usage of the found binary incompatible APIs within the client. These queries are based on Eqs. 6 and 7 in Section 4.3.

Findings and discussion Figure 29 shows an example of a bug[18] reported in Eclipse Orbit.[19] Orbit depends on ASM,[20] a Java bytecode manipulation library. ASM introduced breaking changes in its later releases, such as ClassVisitor being changed from an interface (version 3.X) to a class in version 4.0. This change is a major change in the API and therefore breaking the older 3.X API releases.

We illustrate how our ontology-based API dependency measures can aid developers in detecting and dealing with such breaking changes. For the analysis, we extract and populate facts about the breaking changes between different versions of ASM releases and the source code of all projects which depend on ASM releases (8109 dependencies in total). Based on the

[18] https://dev.eclipse.org/mhonarc/lists/cross-project-issues-dev/msg10487.html
[19] https://www.eclipse.org/orbit/
[20] http://asm.ow2.org/

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

Fig. 27 SPARQL query for inferring the BCI_{direct} measure in a project

extracted source code and dependency information, the earlier introduced SPARQL queries can now be used to identify the potential direct and indirect impacts of ASM breaking changes on client applications.

Figure 30 shows the distribution of (a) breaking changes, (b) non-breaking changes, and (c) breaking change densities (BCD) across all selected 20 ASM releases. Figure 30d reports on the impact of the ClassVisitor API breaking change on client applications. Furthermore, this particular change can potentially affect on average 50 different API elements and as many as 225 elements in a single client application. The reported impact set returned by our approach would include clients which reuse the ClassVisitor API either directly (through an implementation of the interface) or indirectly (through transitive inheritance or method invocations).

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

            ?dependency2 code:containsCodeEntity ?priorAPIElement.
            FILTER(?dependency1 != ?dependency2).
          }
        }
      }

    }

}}
```

Fig. 28 SPARQL query for inferring the $BCI_{indirect}$ measure in a project

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

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

Hi

2. org.objectweb.asm

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

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

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

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

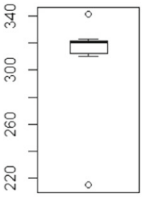
The problem occurs in standalone applications that only have one class loader and so use the first ASM on the classpath. Applications need to tolerate whatever other standalone contributors or legacy startup commands may inflict. Therefore Xtext [1] and OCL [2] are both evolving to tolerate a range of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

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

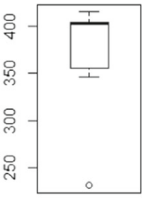
Fig. 29 An example of a reported bug showing how a breaking change in the ASM library impacts Orbit and its dependent projects

5.5 Assessment process

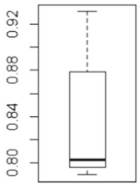
The above sub-sections described how we can identify and measure different attributes of trustworthiness by taking advantage of our unified ontological knowledge representation and



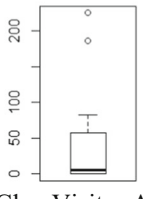
(a) Number of breaking changes



(b) Number of non-breaking changes



(c) BCD of ASM libraries



(d) BCI of the ClassVisitor API in ASM libraries

Fig. 30 Distribution of breaking changes and their impacts in the analyzed ASM libraries

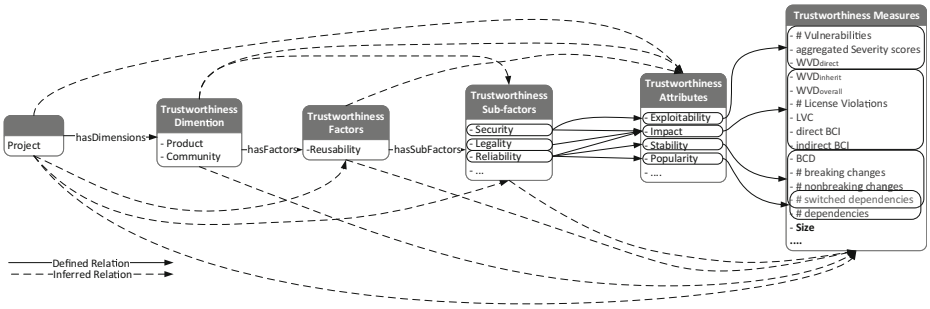


Fig. 31 Overview of relations in the semantic OntTAM domain model

Semantic Web reasoning services. The OntTAM assessment process further integrates these scores across attributes and sub-factors. For the actual assessment process, we first compute the fuzzy score for each measure individually and then aggregate these scores to calculate the attribute, sub-factors, factors, and dimension assessment scores. Figure 31 gives a complete overview of how the sub-factors, attributes, and measures are related and used to derive our trustworthiness assessment.

The effect of the fuzzification on the assessment scores typically increases with assessment abstraction levels (e.g., quality dimension scores vs. attribute scores). Figure 32 shows the rules we used to create the fuzzified score for the WVD measure and Fig. 33 provides example rules we used to combine the fuzzified LVC and WVD scores into a score for the *Impact* attribute.

Using the property chain axioms explained in Section 4.2.2, one can now automatically infer trustworthiness scores from the populated measures of any given project. Figure 34 provides a list of sample queries used for integration and fuzzification.

Findings and discussion Table 9 presents a summary of trustworthiness scores from the three software trustworthiness categories we consider in the scope of this work: API breaking changes, security vulnerabilities, and license violations.

```
FUNCTION_BLOCK WVD

VAR_INPUT
   WVD_Measure: REAL;
   WVD_Weight: REAL;
END_VAR

VAR_OUTPUT
   WVD_Score: REAL;
END_VAR

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

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

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

   METHOD : COG;
END_DEFUZZIFY
```

```
RULEBLOCK WVD_SCORE_RULES

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

END_RULEBLOCK

END_FUNCTION_BLOCK
```

Fig. 32 Sample FCL file for creating fuzzy scores for the WVD measure

```
RULEBLOCK IMPACT _SCORE_RULES
RULE 0 : IF LVC_Score IS EXCELLENT AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS AVERAGE ;
RULE 1 : IF LVC_Score IS VERYGOOD AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS POOR ;
RULE 2 : IF LVC_Score IS AVERAGE AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS POOR ;
RULE 3 : IF LVC_Score IS POOR AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS VERYPOOR ;
RULE 4 : IF LVC_Score IS VERYPOOR AND WVD_Score IS VERYPOOR THEN   IMPACT_Score IS VERYPOOR;
…
END_RULEBLOCK
END_FUNCTION_BLOCK
```

Fig. 33 Sample FCL file for integrating the LVC and WVD fuzzy scores for the Impact attribute

```
Query 1: At sub-factor level
SELECT distinct ?project ?subfactorScore
WHERE {
  ?impactAttribute a onttam:SubFactor.
  ?project onttam:hasSubfactor ?subfactorAttribute.
  ?subfactorAttribute onttam:hasScore  ?subfactorScore.
}

Query 2: At factor level
SELECT distinct ?project ?factorScore
WHERE {
  ?factorAttribute a onttam:Factor.
  ?project onttam:hasFactor ?factorAttribute.
  ?factorAttribute onttam:hasScore  ?factorScore.
}
```

Fig. 34 SPARQL query illustrating the inference of overall trustworthiness scores

Tables 10 and 11 show the results of our queries in Fig. 34 for our studied projects and demonstrate the effectiveness of our OntTAM model. The results indicate that despite the presence of security, licensing, and breaking change concerns, almost all projects have excellent trustworthiness scores at the presented sub-factor and factor levels. This is due to how the score categories are distributed over the fuzzy scale. In our work, the categories are distributed equally from 0 to maximum measure value recorded in our dataset. For example, the maximum WVD measure in our dataset is 11.29, making all WVD measures under 2.95 excellent. The complete scale distributions for all our measures can be found in the FCL files online.[21]

It should be noted that the tables do not report on the final overall trustworthiness score since this score would require a particular assessment context and an instantiation of our OntTAM assessment model with more measures, attributes, and sub-factors, which we omitted in this study due to space limitations.

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

Table 9 Overview of selected trustworthiness measure scores

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

6 Related work

6.1 Library recommendation and migration techniques

Many third-party libraries are available for download to reduce development time by providing access to features ready for use. To help developers take advantage of these libraries, several techniques have been proposed that provide automatic library recommendations to developers. Common to these approaches is that they rely on criteria such as popularity and stability. Some of them even rely on the client's context (e.g., mining previous usage of libraries) for their recommendations. For example, Mileva et al. (2010) study the popularity of an API. Their approach studies the rate at which dependencies adopt or switch from OSS libraries. Hora and Valente (2015) build on Mileva's approach to introduce four distinct API popularity trends:

Table 10 Example of inferred trustworthiness scores at sub-factor level

Project	Security SubFactor		Legality SubFactor		Reliability SubFactor	
	Numerical score	Fuzzified score	Numerical score	Fuzzified score	Numerical score	Fuzzified score
commons-fileupload 1.0	5.01	Average	1.45	Excellent	0	Excellent
commons-fileupload 1.1	5.01	Average	1.45	Excellent	0	Excellent
commons-fileupload 1.2	5.01	Average	1.45	Excellent	0	Excellent
commons-fileupload 1.2.1	1.45	Excellent	1.45	Excellent	0	Excellent
commons-fileupload 1.2.2	1.45	Excellent	1.45	Excellent	0	Excellent
commons-fileupload 1.3	3.77	VeryGood	1.45	Excellent	0	Excellent
Apache CXF WS Security 2.4.1	1.45	Excellent	1.45	Excellent	0	Excellent
Apache CXF WS Security 2.4.4	1.45	Excellent	1.45	Excellent	0	Excellent
Apache CXF WS Security 2.4.6	1.45	Excellent	1.45	Excellent	0	Excellent
Apache CXF WS Security 2.6.3	1.45	Excellent	1.45	Excellent	0	Excellent
Apache CXF WS Security 2.7.0	1.45	Excellent	1.45	Excellent	0	Excellent
Struts 1.2.4	1.45	Excellent	1.45	Excellent	0	Excellent
Struts 1.2.8	1.45	Excellent	1.45	Excellent	0	Excellent
Struts 1.2.9	1.45	Excellent	1.45	Excellent	0	Excellent

Table 11 Example of inferred trustworthiness scores at factor level

Project	Reusability factor	
	Numerical score	Fuzzified score
commons-fileupload 1.0	0	Excellent
commons-fileupload 1.1	0	Excellent
commons-fileupload 1.2	0	Excellent
commons-fileupload 1.2.1	1.45	Excellent
commons-fileupload 1.2.2	1.45	Excellent
commons-fileupload 1.3	1.45	Excellent
Apache CXF WS Security 2.4.1	1.45	Excellent
Apache CXF WS Security 2.4.4	1.45	Excellent
Apache CXF WS Security 2.4.6	1.45	Excellent
Apache CXF WS Security 2.6.3	1.45	Excellent
Apache CXF WS Security 2.7.0	1.45	Excellent
Struts 1.2.4	1.45	Excellent
Struts 1.2.8	1.45	Excellent
Struts 1.2.9	1.45	Excellent

fast growth, constant growth, peak growth, and dead growth. Their approach is shown to be of benefit to both library developers and clients. For example, library developers can be notified when the popularity of their API begins to go down. Raemaekers et al. (2012) present four stability metrics that calculate the stability of API interfaces. They demonstrate how the metrics can be used by developers in deciding on libraries to reuse. The frequency of the migration of API dependencies has also been used to determine the stability of an API by Teyton et al. (2012), Hora and Valente (2015), and Mileva et al. (2009).

Other techniques exist which recommend various API elements (method calls, blocks of code, etc.) of a software library to developers using heuristics that leverage various information sources (source code, commit logs, etc.). Thung et al. (2013) propose an automated technique, which combines association rule mining techniques and collaborative filtering to perform the recommendation of libraries. Their approach recommends a number of likely relevant libraries to developers of a target project based on the libraries used by other projects. McCarey et al. recommend methods of software libraries to a developer by investigating the history of methods that have been used in the past (McCarey et al. 2005).

In addition, several API documentation and tutorial analysis approaches have been introduced to aid developers in understanding how features provided by software libraries can be correctly utilized. For example, Jiang et al. (2017) introduced an unsupervised machine learning approach, which identifies and recommends parts of API tutorials that are relevant to a developer's API usage context. Maalej and Robillard (2013) analyzed API reference documentation, to identify and report on patterns of knowledge found in API documentation, with the goal to help practitioners evaluate and organize the content of their API documentation.

The abovementioned techniques mostly focus on the use reuse of external libraries. Our work aims to provide developers with an approach to assess how much trust can be placed on a recommended software library. Our work can be seen complementary to such existing library recommendation systems, in terms of extending these existing recommendation criteria by making quality in the form of trustworthiness an integrated part of the library recommendations.

6.2 Impact of API breaking changes

A significant amount of research exists that studied real-world systems and how they are impacted by API changes and the ripple effects of these changes (Raemaekers et al. 2012; Raemaekers et al. 2014; Robbes et al. 2012; Cossette and Walker 2012; Kapur et al. 2010). Many approaches have been proposed that apply impact analysis in order to mitigate change impacts on client applications. Dig and Johnson (2006) define a catalog of breaking and non-breaking changes. They observed that refactoring accounts for 80% of the changes that break client systems. Raemaekers et al. (2012) present four stability metrics based on method changes and removals. The authors investigate their metrics behavior by performing a historical analysis of stability and impact on 140 clients of the Apache Commons Library. Jezek et al. (2015) use a dataset of 109 Java programs and 564 program versions to analyze binary compatibility in the context of OSGi-based systems. Xavier et al. (2017) conducted a large empirical study on 317 real-world Java libraries, 9K releases, and 260K client applications to investigate the impact of API breaking changes on client applications. Decan et al. observed that about 1 in every 20 updates to a CRAN package was a backward incompatible change, accounting for 41% of the errors in released packages that depended on them (Decan et al. 2016).

Complicated and changing dependencies are difficult to identify by developers (Artho et al. 2012) and have led to common expressions like “DLL hell” and “dependency hell.” In our work, we extend the scope of our impact analysis of breaking changes to a more global scope, in which the analysis goes beyond individual project boundaries.

6.3 Software security vulnerability identification

Different approaches for static vulnerability analysis and detection in source code exist, such as Plate et al. (2015) who proposed a technique that supports the impact analysis of vulnerability based on code changes introduced by security fixes. Nguyen et al. (2016) introduced an automated method to identify vulnerable code based on older releases of a software system. Cadariu et al. (2015) introduce in their Vulnerability Alert Service (VAS) an approach that notifies users if a vulnerability is reported for software systems. In contrast to this existing research, we provide a more holistic approach that supports an intra- and inter-project dependency analysis. We take advantage of semantic reasoning services to infer implicit facts about vulnerable code usages within the system and support bi-directional dependency analysis—which also includes both impacts to external dependencies and vice versa. In addition, our analysis results become an integrated part our knowledge base to be reused for different analysis tasks.

6.4 Software license violation identification

Related studies into identifying software license violations can be categorized into two levels: intra-project and inter-project.

At the intra-project level, studies aim to identify the introduction of license violations introduced by having project files with different licenses. Di Penta et al. (2010) proposed an approach to automatically track the licensing evolution of systems, identifying changes in licenses and copyright years. They found that OSS projects do change licenses over time and these changes were not just to new versions of the existing license. Sometimes,

projects that switched licenses altogether had intended and unintended effects on downstream users of these projects. As recently as 2015, research has been conducted by Wu et al. (2015) on the evolution of the licenses specified in the header of each file, with the explicit goal of finding license inconsistencies. They categorize the evolution of licenses as a license addition/removal, upgrade/downgrade, or change. These categorizations are then used to judge whether the new modification/evolution of the license results in an inconsistency.

Zhong and Mei (2017) have shown in their work that developers typically combine APIs from different libraries to solve development problems. However, while such API reuse reduces the coding effort, it also increases substantially the effort required in identifying license violations at the inter-project level. Several researchers have studied how code reuse (through cloning) and software components/libraries can lead to the introduction of license violations. Using code clones to detect small-scale license violations has been investigated for example by Monden et al. (2011). They introduced three quality metrics for code clone detection based on license violations. However, as part of their evaluation, they did not detect any actual license violations in OSS. Instead, they used license violations were merely used as a theoretical use case for their comparative study. The Binary Analysis Tool (BAT) developed by Hemel et al. (2011) detects code clones of OSS in proprietary binaries for the express purpose of finding violations of popular GPL projects. The authors used the comparison of string literals, data compression, and binary deltas. Interestingly, BAT does find many true code clones but falls short by leaving the verification as a manual process, i.e., whether a code clone is also a license violation.

The work by German and Hassan (2009) is the most closely related to our work. The authors created a "model to describe licenses and the implications of licenses on the reuse of components." Their model describes what usage scenarios result in a derived work or not. Our work builds upon the existing body of knowledge for license violations, by providing the first attempt to create a formal representation of the license dependencies. The advantage of our ontological representation, being an integrated part of our unified knowledge model, is the ability to extend and reuse our license model for different types of analysis tasks, such as its seamless integration in a trustworthiness assessment model.

6.5 Quality models

Assessing quality to improve the evolution of software systems has been addressed in existing research through the introduction of software quality models. These models introduced quality dimensions and classified quality factors that affect the development and maintenance of software products. Among the most widely accepted quality assessment model is the ISO 9126[22] software quality model standard which defines a quality model via a set of quality characteristics and sub-characteristics that were believed to be the more representative and relevant at the time of its introduction. As the complexity and vulnerability of software systems grows as a result of their components being increasingly reused across project boundaries and interconnected through networks and communication links, assessing the trustworthiness of systems and their components plays an ever-increasing role. While security and

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

interoperability are already present in the ISO 9126 standard as "sub-factors" of functionality, more recent quality models such as the ISO 25000 standard have extended the ISO 9126, by making security and interoperability a main quality aspect of the standard.

In Hmood et al. (2012), the authors introduced an SE-Evolvable QUality Assessment Meta-model (SE-EQUAM), a quality assessment model which is both evolvable and reusable. The model introduces a set of complementary core requirements necessary for a model to be considered an evolvable model: *Model Reusability*, *Knowledge Modeling*, *Knowledge Population*, and *Knowledge Exploration* (Hmood et al. 2012). In this work, we adopt the model evolvability criteria to derive our trustworthiness meta-model that is not only capable of dealing with continuous change (in the model) but also allows for its reuse by simplifying the instantiation of new domain model instances.

6.6 Trustworthiness models

Existing work on assessing the trustworthiness of OSS systems, for example, Taibi (2008), Larson and Miller (2005), and Tan et al. (2008), have attempted to quantify OSS trustworthiness of software systems in situ, but results are limited to artifacts in the development environment; external and heterogeneous knowledge sources are not considered in these approaches. Other researchers Pfleeger (1992) and Yang et al. (2009) seek to analyze and predict aspects of trustworthiness during software development, while other work has focused on introducing new evaluation criteria to better capture the nature of OSS's components, for example, the QualiPSo model of OSS trustworthiness (del Bianco et al. 2009), and Boland et al. (2010) quantify and assess risk based on the Structured Assurance Case Model (SACM) (Rhodes et al. 2010) to determine software trustworthiness. The main objective of these models is to apply their quality (trustworthy) factors to allow for a standardized product comparison across different projects and domains. Most trustworthiness assessment models share a generic structure, template, or frame for assessing software security quality that corresponds to a hierarchy or tree structure with multiple levels and a set of constraints that define the relationship between one level and the next one. However, regardless of the kind of components, these syntactic proposals mainly address and mostly focus on the evaluation criteria and decision-making phases, setting aside the practical problem of how to search for and locate components and to assign suitable information about them (Land et al. 2009). Also, a general concern in most of these models is that they rely on the software product and traditional software lifecycle artifacts. They do not necessarily consider external resources in their assessment such as external vulnerability databases. As a result, there is no consensus on the applicability of these trustworthiness models in industrial practice (Ayala et al. 2013).

While existing proposals for creating such meta-modeling assessment models focus on adopting one or more of these existing quality models in one standard model, this may result in an incomplete or unbalanced assessment, depending on the input. Using a meta-modeling approach can address this challenge by quantifying the trustworthiness of software as a "product" and specifying a domain model that captures and conceptualizes trustworthiness. A *domain model* is a conceptualization of a problem domain in terms of its entities, properties, relationships, and constraints. In software, several domain models exist that are capable of representing and assessing predefined sets of trustworthiness, e.g., PAS 754:2014, QualiPSo (del Bianco et al. 2009), and Boland et al. (2010). All these domain models share a common,

while informal (non-machine-readable), structural representation of the trustworthiness they are assessing. This lack of formalism and semantics limits the possible reuse and instantiation for specific trustworthiness assessment contexts.

7 Threats to validity

7.1 Internal threats

Design quality One of the major benefits of our approach is the ability to seamlessly integrate and reuse ontologies while ensuring the quality of the resulting knowledge model. Assessing the quality of our ontology designs is an inherently difficult problem since what constitutes quality depends on different non-functional requirements (e.g., reuse, usability, extensibility, expressiveness, and reasoning support). We partly address this threat by using existing reasoners (such as Pellet, Hermit, and JFact) and tools (OOPS![23] and the Neon Toolkit[24]) to check our ontology design for taxonomic, syntactical, and consistency problems. To determine if our ontology constraints were sufficient to identify incorrect data, we incrementally populated the ontologies with facts during the evaluation process. While the reasoners did not report any inconsistencies in our ontologies, OOPS! reported a few problems in our ontologies which violated some of the design rules in OOPS! rule catalog. However, the identified violations were a result of missing license information and annotations (such as rdfs:label and rdfs:comment) for some of our ontology elements.

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

Completeness A potential threat to our approach is whether the set of measures we considered in our assessment as part of OntTAM evaluation is sufficient to capture reusability as a trustworthiness factor. We addressed this threat by selecting our trustworthiness measures from a well-established subset of existing trustworthiness models, such as PAS 754:2014, QualiPSo (del Bianco et al. 2009), and Boland et al. (2010). While we only selected a very small subset of these trustworthiness attributes, we believe this subset is sufficient to illustrate the applicability of our assessment model. In particular, the objective of our study was not to verify the assessment model for its completeness but rather to illustrate that OntTAM can be instantiated to a given (user specified) assessment context. The study shows that instantiating and extending OntTAM to support other requirements including new measures, attributes, or sub-factors is a straightforward task.

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

7.2 External threats

All systems are written in Java The application-level examples described in this paper are limited in their scope to open-source Java projects in the Maven repository, and the results obtained might not be applicable to other programming languages or build repositories. Given that our modeling approach is based on different levels of abstraction, we also abstract common aspects of source code and build dependencies in our knowledge model. We do model the domain of object-oriented programming languages, software vulnerabilities, and software licenses, and build repositories as individual domains of discourse in the domain-specific layer of our knowledge model.

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

8 Conclusion and future work

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. However, the reuse and therefore dependency on such external libraries can affect a project's overall quality through security vulnerabilities, license violations, and breaking changes. In order to address some of these challenges, we introduce OntTAM a trustworthiness assessment model which is an instantiation of our SE-EQUAM assessment model. OntTAM takes advantage of our existing unified knowledge representation of different SE knowledge resources and extends these knowledge bases to allow for an automated analysis and assessment of trustworthiness quality attributes. We argue that ontologies not only promote and support the conceptual representation of knowledge resources in software ecosystems but also allow us to take advantage of semantic reasoning during the assessment of trustworthiness quality factors. We further present a concrete instantiation of our assessment model that not only provides a formal modeling of trustworthy quality attributes but can also be extended/customized to specific stakeholder needs. We illustrate how a concrete instantiation of OntTAM for a small subset of sub-factors, attributes, and measures related to the trustworthiness of reusable components can be created. The measures which we include in the study are API breaking changes, security vulnerabilities, and license violations.

As part of our future work, the assessment process presented in this paper could be extended in several ways. Enriching existing online software repositories with trustworthiness scores to allow and automate the recommendation of trustworthy APIs directly in a programmer's IDE would be one avenue of future work. Also, a detailed user study to qualitatively validate the applicability of our model in actual project contexts is needed.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

Alqahtani, S. S., Eghan, E. E., & Rilling, J. (2016). SV-AF—a Security Vulnerability Analysis Framework, in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 219–229.

Alqahtani, S. S., Eghan, E. E., & Rilling, J. (2017). Recovering semantic traceability links between APIs and security vulnerabilities: an ontological modeling approach. *10th IEEE International Conference on Software Testing, Verification and Validation*.

Artho, C., Suzaki, K., Di Cosmo, R., Treinen, R., Zacchiroli, S., & A. P. S. Distributions (2012). Why do software packages conflict?, 141–150.

Atkinson, C., Gutheil, M., & Kiko, K. (2006). On the relationship of ontologies and models. *Proc. 2nd Work. MetaModelling Ontol. WoMM06 LNI P96 Gesellschaft fur Inform. Bonn*, 47–60.

Ayala, C., Franch, X., Conradi, R., Li, J., & Cruzes, D. (2013). *Developing software with open source software components. Finding source code on the web for remix and reuse* (pp. 167–186). New York: Springer New York.

Bergel, A., Denier, S., Ducasse, S., Laval, J., Bellingard, F., Vaillergues, P., Balmas, F., & Mordal-Manet, K. (2009). SQUALE—Software QUALity Enhancement. *2009 13th European Conference on Software Maintenance and Reengineering*, 285–288.

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American, 284*(5), 34–43.

Boland, T., Cleraux, C., & Fong, E. (2010). *Toward a preliminary framework for assessing the trustworthiness of software* (pp. 1–31). Gaithersburg: National Institute of Standards TechnologyInteragency/Internal Report, U.S. Department of Commerce.

Cadariu, M., Bouwers, E., Visser, J., & Van Deursen, A. (2015). Tracking known security vulnerabilities in proprietary software systems. *2015 IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2015 - Proc*, 516–519.

Cingolani, P., & Alcala-Fdez, J. (2012). jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation. *2012 IEEE International Conference on Fuzzy Systems*, 1–8.

Cossette, B. E. & Walker, R. J. (2012). Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng, 55:1–55.

Decan, A., Mens, T., Claes, M., & Grosjean, P. (2016). When GitHub meets CRAN: an analysis of inter-repository package dependency problems. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 493–504.

Di Penta, M., German, D. M., Guéhéneuc, Y.-G., and Antoniol, G. (2010). An exploratory study of the evolution of software licensing, Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. - ICSE '10, vol. 1, p. 145.

Dig, D., & Johnson, R. (2006). How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice, 18*(2), 83–107.

DuCharme, B. (2011). *Learning SPARQL* (2nd ed.). Sebastopol: O'Reilly Media.

F. S. Foundation (2014). Various licenses and comments about them. *GNU Project* [Online]. Available: https://www.gnu.org/licenses/license-list.en.html. Accessed 22 July 2017.

Gao, J. Z., Chen, C., Toyoshima, Y., & Leung, D. K. (1999). Engineering on the Internet for global software production. *Computer (Long. Beach. Calif)., 32*(5), 38–47.

German, D. M. & Hassan, A. E., (2009). License integration patterns: addressing license mismatches in component-based development. *2009 IEEE 31st International Conference on Software Engineering*, 188–198.

Hemel, A., Kalleberg, K. T., Vermaas, R., & Dolstra, E. (2011). Finding software license violations through binary code clone detection. *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, 63–72.

Henderson-Sellers, B. (2011). Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software, 84*(2), 301–313.

Hmood, A., Schugerl, P., Rilling, J., & Charland, P. (2010). OntEQAM—a methodology for assessing evolvability as a quality factor in software ecosystems. *Defence R&D Canada - Valcartier, Valcartier QUE (CAN)*, 8.

Hmood, A., Keivanloo, I., & Rilling, J. (2012). SE-EQUAM—an evolvable quality metamodel. *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, 334–339.

Hora, A. & Valente, M. T. (2015). apiwave: keeping track of API popularity and migration, 321–323.

I. E. Commission (2000). Programmable controllers—part 7: fuzzy control programming.

Jezek, K., Dietrich, J., & Brada, P. (2015). How Java APIs break—an empirical study. *Information and Software Technology, 65*, 129–146.

Jiang, H., Zhang, J., Ren, Z., & Zhang, T. (2017). An unsupervised approach for discovering relevant tutorial fragments for APIs. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 38–48.

Kagdi, H., Yusuf, S., & Maletic, J. I. (2006). Mining sequences of changed-files from version histories. *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, 47.

Kagdi, H., Collard, M. L., & Maletic, J. I. (2007). Comparing approaches to mining source code for call-usage patterns. *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 20–26.

Kamiya, T., Kusumoto, S., & Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering, 28*(7), 654–670.

Kapur, P., Cossette, B., & Walker, R. J. (2010). Refactoring references for library migration. *ACM SIGPLAN Notices, 45*(10), 726.

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

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

del Bianco, V., Lavazza, L., Morasca, S., & Taibi, D. (2009). Quality of open source software: the QualiPSo trustworthiness model, 199–212.

Land, R., Sundmark, D., Lüders, F., Krasteva, I., & Causevic, A. (2009). Reuse with software components—a survey of industrial state of practice. *Form. Found. Reuse Domain Eng*, 150–159.

Larson, D., & Miller, K. (2005). Silver bullets for little monsters: making software more trustworthy. *IT Prof., 7*(2), 9–13.

Maalej, W., & Robillard, M. P. (2013). Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering, 39*(9), 1264–1282.

Mann, C. J. H. (2003). The description logic handbook—theory, implementation and applications. *Kybernetes, 32*(9/10), k.2003.06732iae.006.

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). Factors in software quality. Volume I. Concepts and definitions of software quality.

McCarey, F., Cinnéide, M. Ó., & Kushmerick, N. (2005). Rascal: a recommender agent for agile reuse. *Artificial Intelligence Review, 24*(3–4), 253–276.

McGuinness, D. L. and Van Harmelen, F. (2004). Owl web ontology language overview. W3C Recomm. 10.2004–03, 2004, 1–12.

Mileva, Y. M., Dallmeier, V., Burger, M., & Zeller, A. (2009). Mining trends of library usage. *Proc. Jt. Int. Annu. ERCIM Work. Princ. Softw. Evol. Softw. Evol,* 57–62.

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

Monden, A., Okahara, S., Manabe, Y., & Matsumoto, K. (2011). Guilty or not guilty: using clone metrics to determine open source licensing violations. *IEEE Software, 28*(2), 42–47.

Nguyen, V. H., Dashevskyi, S., & Massacci, F. (2016). An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering, 21*(6), 2268–2297.

Parnas, D. L. (1994). Software aging. *ICSE '94 Proceedings of the 16th international conference on Software engineering*, 279–287.

Pfleeger, S. L. (1992). Measuring software reliability. *IEEE Spectrum, 29*(8), 56–60.

Plate, H., Ponta, S. E., & Sabetta, A. (2015). Impact assessment for vulnerabilities in open-source software libraries. *2015 IEEE 31st Int. Conf. Softw. Maint. Evol. ICSME 2015 – Proc,* 411–420.

Raemaekers, S., Van Deursen, A., & Visser, J. (2012). Measuring software library stability through historical version analysis. *IEEE Int. Conf. Softw. Maintenance, ICSM*, 378–387.

Raemaekers, S., Van Deursen, A., & Visser, J. (2014). Semantic versioning versus breaking changes: a study of the maven repository. *Proc. - 2014 14th IEEE Int. Work. Conf. Source Code Anal. Manip. SCAM 2014*, 215–224.

Rahman, M. M., Roy, C. K., & Lo, D. (2016). RACK: automatic API recommendation using crowdsourced knowledge. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 349–359.

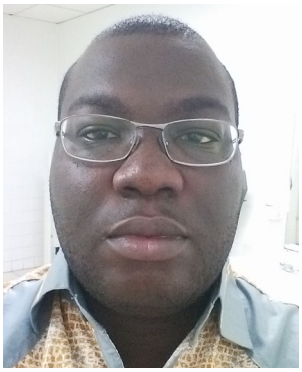
Rhodes, T., Boland, F., Fong, E., & Kass, M. (2010). Software assurance using structured assurance case models. *Journal of Research of the National Institute of Standards and Technology, 115*(3), 209–216.

Robbes, R., Lungu, M., & Röthlisberger, D. (2012). How do developers react to API deprecation?. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, 1.

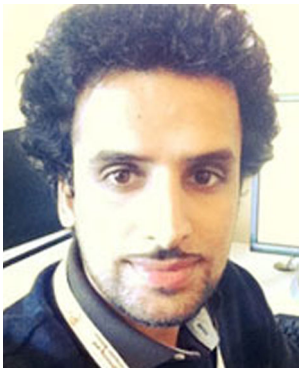
Samoladas, I., Gousios, G., Spinellis, D., & Stamelos, I. (2008). The SQO-OSS quality model: measurement based open source software evaluation. *Open Source Development, Communities and Quality*, Boston, MA: Springer US, 237–248.

Seedorf, S. & Mannheim, F. F. I. U. (2006). Applications of ontologies in software engineering. *In 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*.

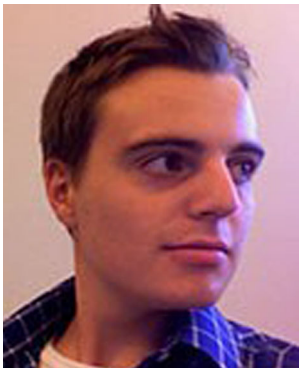
Seneviratne, O., Kagal, L., Weitzner, D., Abelson, H., Berners-Lee, T., & Shadbolt, N. (2009). Detecting creative commons license violations on images on the world wide web. *WWW2009*.
Taibi, D. (2008). Defining an open source software trustworthiness model. *Proc 3rd Int Dr Symp Emperical Software Eng*, 4.
Tan, T., He, M., Yang, Y., Wang, Q., & Li, M. (2008). An analysis to understand software trustworthiness. *2008 The 9th International Conference for Young Computer Scientists*, 2366–2371.
Teyton, C., Falleri, J. R., & Blanc, X. (2012). Mining library migration graphs. *Proceedings of Work. Conf. Reverse Eng. WCRE*. 289–298.
Thung, F., Lo, D., & Lawall, J. (2013). Automated library recommendation. *Proceedings of Workshop Conference on Reverse Engineering. WCRE*, 182–191.
Williams, J., & Dabirsiaghi, A. (2012). *The unfortunate reality of insecure libraries* (pp. 1–26). Appleton: Asp. Secur. Inc.
Witte R., Zhang Y., & Rilling J. (2007). Empowering software maintainers with semantic web technologies. *ESWC, 4519*, 37–52.
Wu, Y., Manabe, Y., Kanda, T., German, D. M., & Inoue, K. (2015). A method to detect license inconsistencies in large-scale open source projects. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 324–333.
Würsch, M., Ghezzi, G., Hert, M., Reif, G., & Gall, H. C. (2012). SEON: a pyramid of ontologies for software evolution and its applications. *Computing, 94*(11), 857–885.
Xavier, L., Brito, A., Hora, A., & Valente, M. T., (2017). Historical and impact analysis of API breaking changes: a large-scale study. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 138–147.
Yang Y., Wang Q., & Li M. (2009). Process trustworthiness as a capability indicator for measuring and improving software trustworthiness. *ICSP, 5543*, 389–401.
Zadeh, L. A. (1975). The concept of a linguistic variable and its application to approximate reasoning-III. *Information Sciences, 9*(1), 43–80.
Zhang, Y., Witte, R., Rilling, J., & Haarslev, V. (2008). Ontological approach for the semantic recovery of traceability links between software artefacts. *IET Software, 2*(3), 185.
Zhong, H. & Mei, H. (2017). An empirical study on API usages. *IEEE Trans. Softw. Eng. (Early Access)*, 1.



Ellis E. Eghan is currently a Ph.D. candidate in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Prior to joining the Ph.D. program at Concordia, he received a Master in Applied Computer Science from Concordia University and a Bachelor degree in Computer Science from Kwame Nkrumah University of Science and Technology in Ghana. His research is mainly focused on improving and supporting software engineering tasks using semantic analysis of software build systems through semantic web techniques. He has currently published three papers in major refereed international journals and conferences.



Sultan S. Alqahtani is currently a Ph.D. candidate in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Prior to joining the Ph.D. program at Concordia, he received a Master in Information Systems Security from Concordia Institute for Information Systems Engineering (CIISE) in Montreal, Canada, and a Bachelor degree in Computer Science from Imam Mohammad bin Saud University in Riyadh, Saudi Arabia. Sultan's research as a member of the Ambient Software Engineering Group (ASEG) is mainly focused on improving and supporting software engineering tasks using semantic analysis of software security vulnerability databases through semantic web technologies. Sultan's contributions published so far during his Ph.D. research five articles in major refereed international journals and conferences.



Christopher Forbes is currently a Master student in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Prior to joining the Master's program at Concordia, he received a Bachelor degree in Computer Science from University of Ontario Institute of Technology. His research is mainly focused on improving and supporting software engineering tasks using semantic web techniques. He has currently co-published four papers in major refereed international journals and conferences.

Dr. Juergen Rilling is a Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He obtained a Diploma degree in Computer Science from the University of Reutlingen, Germany, in 1991 and a M.Sc. in Computer Science from the University of East Anglia, UK, in 1993. He received his Ph.D. from the Illinois Institute of Technology, Chicago, USA, in 1998. The general theme of his research over the last 19 years has been on providing software maintainers with techniques, tools, and methodologies to support the evolution of software systems. His current research focus is on supporting the modeling and analysis of global software ecosystems. He has published over 100 papers in major refereed international journals, conferences, and workshops. Dr. Rilling also serves on the program committees of numerous international conferences and workshops in the area of software maintenance and program comprehension and as a reviewer for all major journals in his research area.