11

Enriching SE Ontologies with Bug Quality

Philipp Schugerl , Juergen Rilling , Philippe Charland

11.1 Motivation

Semantic Web technologies have had a major impact on how we perceive available information on the Internet. Instead of seeing data spread across different locations as isolated information silos, the Semantic Web allows us to connect the dots. This is specifically important in software engineering, where many systems have been left so far in isolation or are only loosely connected. For example, a modification request is initiated by an issue tracker ticket, with a maintenance process being followed as part of the change request and commits to a version control system. Nevertheless, each system (issue tracker, time and process management, version control) tends to be only loosely connected with each other. This leaves developers with limited possibilities to query and trace information across these systems or automatically detect problems across system boundaries. Being able to establish such links between the involved systems and artefacts is a key requirement in many engineering disciplines (e.g., civil and manufacturing). In order for software engineering to mature like other engineering disciplines, it is essential to model and connect these systems to leverage the available information. Furthermore, this will also provide the basis for improved tool support for developers and maintainers to explore and reason upon information in these systems. While traditional approaches to unify software engineering systems (e.g., Rational Team Concert) are prone to fail due to their need to compete against the features of specialized applications, semantic technologies, in contrast, allow the creation of a non-intrusive layer above these individual systems.

Semantic technologies have long been applied for conceptualizing various aspects of the software engineering domain, with the objective to reduce both the abstraction and semantic gap among artefacts such as source code, bug tracking and version control systems. Nevertheless, these rather simple domain models, as they can often be found in today's state of the art software engineering ontologies, are not the silver bullet for addressing the fundamental challenges. While helpful in establishing a conceptualization of the prob-

lem domain and providing basic traceability among artefacts, more advanced ontologies need to be tailored to support specific usage scenarios. This is specifically true whenever reasoning services are applied or required, which impose additional constraints on the ontology design. Furthermore, ontology design is also constrained by the capabilities of the reasoner itself, since not every problem (e.g., complex mathematical functions or other computationally intensive tasks) might be suitable to be modelled in description logic or supported efficiently by a particular reasoner.

11.2 State of the art

As a knowledge representation language, *Web Ontology Language (OWL)* has already been applied in many applications of the software engineering domain, such as model-driven software development [44], reverse engineering tool integration [27], and component reuse [25]. There exists relevant work on conceptualizing the software engineering domain to support the teaching of software engineering, e.g., [3]. Petrenco et al. [35] used open source software systems in teaching software evolution. Their experience showed that by integrating a software change process model into course projects and by collecting feedback, it was possible to enhance the assessment of student performance. Falbo et al. [20] reported on shared conceptualization for integrated tool development, and Deridder et al. [17] have used ontologies for linking artefacts at several phases of the development process. The SWEBOK project [41] applies ontologies in software engineering to provide pointers to relevant literature on each of its concepts. Current web-based learning approaches [36] focus on reusability in their content design. Wongthongtham et al. [47] [8] introduced a software engineering ontology for the collaborative nature of software engineering. Ankolekar et al. [4] modelled bugs and software components using an ontology. However, their presented model does not take advantage of any reasoning services and lacks the ability to generate user specific advice. Common to all of these approaches is their main intent to support, in one form or another, the conceptualization of knowledge, mainly by standardizing the terminology for knowledge sharing based on a common understanding. These approaches typically fall short on adopting and formalizing a process model that supports connecting knowledge resources in a knowledge base (KB) with process activities. In contrast to the work in this chapter, the mentioned approaches do not integrate collaborative and ambient software engineering. They also lack the ability to establish context-awareness through the use of reasoning services to infer implicit knowledge that provides contextual guidance.

In terms of source code representations in ontologies, the *FAMOOS Information Exchange Model (FAMIX)* tree representation [16] is a source code meta-model that can be used as an exchange format for object-oriented programming languages. It has been used prior to ontologies as a description

language, e.g. [39], where similarities between classes and software projects are measured after modelling an *Abstract Syntax Tree (AST)* in the FAMIX representation. In [19], an OWL ontology of software design patterns is presented. The ontology is used to scan an AST for source code patterns that identify a specific design pattern. A general purpose open source bug tracker ontology is introduced in [1], but no specific implementation that uses the given ontology is presented. In [46], a system called Code Based Management Systems (CBMS) is presented which uses an AST source code representation. Its main focus is the detection of side effects (e.g., erroneously changed global variables).

Other uses of ontologies in software engineering include a CMMI-SW model representation and the use of reasoning for classifying organizational maturity levels [42]. However, only limited research exists on modelling software evolution using ontologies. Ruiz et al. [38] present a semi-formal ontology for managing software maintenance projects. They consider both the static and dynamic aspects, such as the workflow in software maintenance processes. Kitchenham et al. [32] designed a *Unified Modeling Language (UML)* based ontology for software maintenance to identify and model factors that affect the results of empirical studies. Dias et al. [18] extended the work of Kitchenham by applying a first order logic to formalize knowledge involved in software maintenance. Gonzlez-Prez and Henderson-Sellers present a comprehensive ontology for software development [24] that includes a process sub-ontology modelling, among others, techniques, tasks, and workflows. Despite considerable research on ontologies representing functional requirements, little work exists on using ontologies to represent non-functional requirements. More recently, the collaborative nature of software engineering has been addressed by introducing Wiki systems into the software engineering process. Semantic Wiki extensions like Semantic MediaWiki [34] or IkeWiki [40] add formal structuring and querying extensions using the *Resource Description Framework (RDF)* and OWL metadata. Existing work on ontology-based rating models include the modelling and evaluation of service quality to allow consumers locate the quality of service they are looking for [43]. In [15], the QuOnt ontology is presented to codify and conceptualize quality criteria. In [23], a study was conducted to illustrate the use of ontologies in the software measurement domain.

11.3 Bug trackers

Software repositories (such as version control and bug tracking systems) are used to help manage the progress and evolution of software projects. More recently, research has started to focus on identifying ways in which mining these repositories can help software development and evolution. These software repositories contain explicit and implicit knowledge about software projects that can be extracted to provide additional insights to guide continu-

ous software development and plan evolutionary aspects of software projects. In what follows, we focus on the modelling and analysis of bug reports found in bug repositories. Large projects often use bug tracking tools to deal with defect reports. These bug tracking systems allow users to report, track, describe, comment on, and classify bug reports and feature requests. One popular example of such a bug tracking tool commonly found in the open source community is Bugzilla [2].

Bug trackers store error reports in a structured format and therefore offer advanced means to search within them. For example, queries can be submitted to identify all bugs that have been added at a specific point in time or have been associated with a specific component. Bug trackers therefore represent a repository to report, store, and retrieve error reports. While the original purpose of bug tracking systems has been the management of bug reports, their usage meanwhile has shifted to include all kinds of requests such as feature requests, improvements, and general tasks [48]. In many agile approaches, bug trackers even hold a full list of requirements for the final system (e.g., the product backlog in the SCRUM software process). Due to their more general usage, bug trackers are nowadays more appropriately referred as issue trackers.

Bug reports are closely related to versions, revisions, and other software engineering concepts. An issue is reported for a certain version of the software and can either be a bug, a feature/improvement, or a task. It is resolved in one or more revisions that might form another one. Issues may be related to design decisions, requirements, and other parts of the system specifications and documentation. Issues are also closely linked to processes and workflows, as they include a notion of priority that is used to classify the importance (urgency) of the issue to be resolved.

11.4 Objective

The main objective of this chapter is to demonstrate how semantic Web technologies can be applied in software engineering for domain specific tasks, namely on the automated evaluation of various qualities of bug reports. We illustrate how knowledge about the quality of bug reports can guide the knowledge exploration process and be used to manage inconsistent information within an ontology. For the quality assessment of the bug reports we take advantage of the following properties of ontologies:

1. Integrity: Ontologies allow us to check constraints that are not enforced in one or more systems such as the inverse relation between “blocks” and “depends”, or the inconsistency between “platform” and “operating system” in the bug tracker. They also allow to correctly map special properties (e.g., “all” or “unknown”) to respond correctly in queries.

2. Quality classification: A quality classification of bug reports can be automatically performed by the reasoner after enriching the ontology with *Natural Language Processing (NLP)* information. This can be used to improve the quality of their free form description by giving feedback to reporters and support the automated identification of low quality bug reports that often tend to be invalid ones.
3. Traceability and querying: Bug reports are connected to source code and version control ontologies in order to allow a maintainer to automatically discover important links. Queries may leverage information from all three ontologies (e.g., "What methods have been changed with a bug fix").

Existing work on analysing bug reports has shown that many reports in bug repositories contain invalid or duplicate information [6]. For the remaining ones, a significant portion tends to be of low quality, due to the omission of important information, or by adding irrelevant information (noise) to them. As a result, many end up being treated in an untimely or delayed manner. Providing an automated or semi-automated approach to evaluate the quality of bug reports can therefore provide an immediate added benefit to organizations that often have to deal with a large number of bug reports. Similarly, the automatic assignment of maintainers to bugs is an important problem in large projects. The additional integrity constraints checked by the reasoner allow the bug tracker user to make logical assumptions about the provided data and prevent errors during data input. Traceability and querying built in the ontological infrastructure as well as the uniform ontological representation can be used "out of the box" with *Simple Protocol and RDF Query Language (SPARQL)* [29] and today's semantic frameworks.

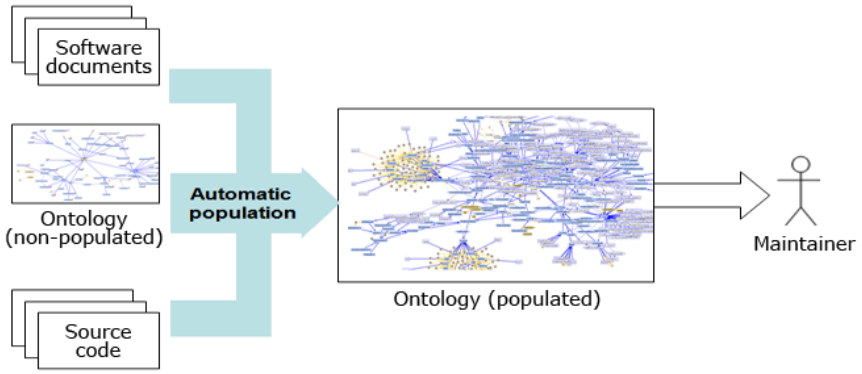

Fig. 11.1. Supporting system evolution through semantic Web technologies

The presented research is part of a larger project on applying semantic Web technologies to support system evolution [37]. The approach is based on

the use of a common formal ontological representation to integrate different software artefacts. Among the artefacts we have modelled and populated in sub-ontologies so far are bug reports, source code repositories, documentation artefacts, and high-level process definitions (Figure 11.1). The ontological representation provides us with the ability to reduce both the abstraction and semantic gap that normally exist among these artefacts. Concepts and their instances, in combination with their relationships, are used to explore and infer explicit and implicit knowledge across sub-ontologies (artefacts). Our semantic web-based software evolution environment does not only support knowledge exploration across artefacts, but also the re-establishment of traceability links among them [7].

11.5 Ontology

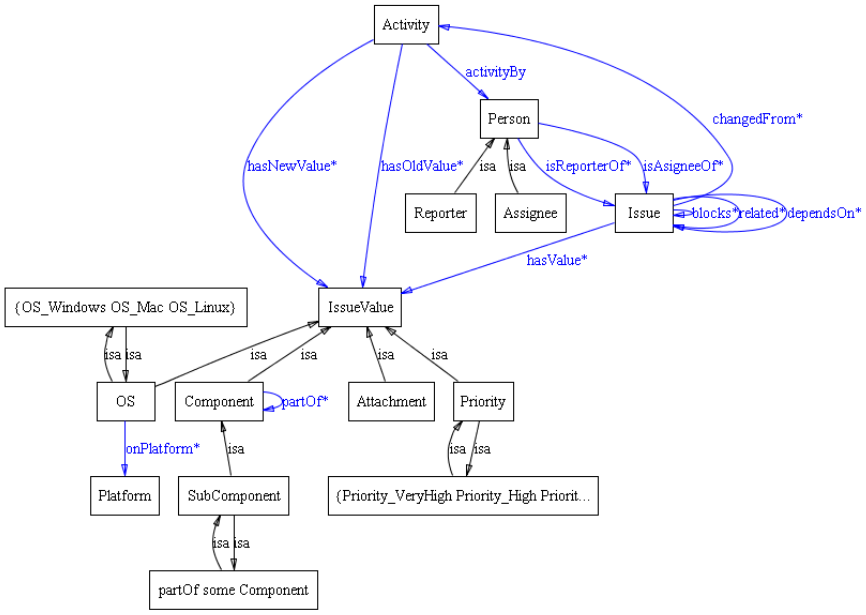

Fig. 11.2. Bug tracker ontology

The ontology defined as part of this chapter models some key concepts that can be found in a bug tracker. Figure 11.2 shows a simplified overview of the Bugzilla ontology with its major concepts and relations. An issue can be changed by an activity and each activity changes issue values such as operating system, component, attachment, or priority. Operating system and priority are

modelled as enumerations. Note that only the static unclassified structure of the ontology is shown. Assignee, for example, is defined as a Person, but it is also an IssueValue.

As part of a bug tracker ontology, various relations among issues are modelled, including their direct and indirect dependencies. In the following example, one issue depends on another one, meaning it cannot be resolved prior to the resolution of the issue it depends on. This is expressed through the "blocks" relation (inverse of "depends on"). By extracting the corresponding fields from a bug tracker repository and populating the ontology, a simple query such as "select open issues which are not blocked by other issues" can be performed. In this context, it has to be noted that information extracted from bug trackers tends to be incomplete and inverse properties or transitive dependencies may not have been included during the original submission by the user. Using an ontological representation, reasoning services can be used to automatically infer these missing relations.

Figure 11.3 illustrates such a simple chain of issues, which are connected by a "depends on" relation. The inverse property can be resolved through the use of reasoning services. Given this internal representation, queries can be formulated using both the "blocks" and "depends on" relation, providing consistent results. Similarly, the transitive relation between issues (e.g., Issue 1 depends on Issue n) can be resolved using reasoning.

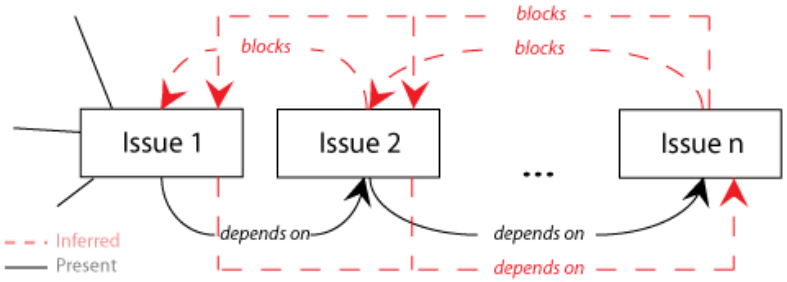

Fig. 11.3. Inferred knowledge within bug tracker ontology

Another example of how the ontology design can help in checking the integrity of the underlying data is through the dependencies between the properties of an issue. Although some issue trackers can model and check dependencies, most use only text properties without further semantic meaning. An example of such properties is the "platform" and "operating system" fields in Issuezilla. While the "operating system" field specifies the system (Linux, Mac, Windows, Solaris, ...) under which the bug occurred, the platform field lists the architecture (PC, Macintosh, Sun, ...) used. There is a clear relation between these two fields, and it only makes sense to combine certain values

with each other (e.g., Solaris with Sun). Although such relations seem trivial, they nevertheless occur as input mistakes and can be filtered out or be used as indicators (e.g., a virtual machine has been used if PC and Mac appear together) to auto complete other properties related to an issue. Ontologies also allow us to easily model more complex relations such as the dependency between "version" and "operating system" (only certain versions of an application might have been released for an operating system).

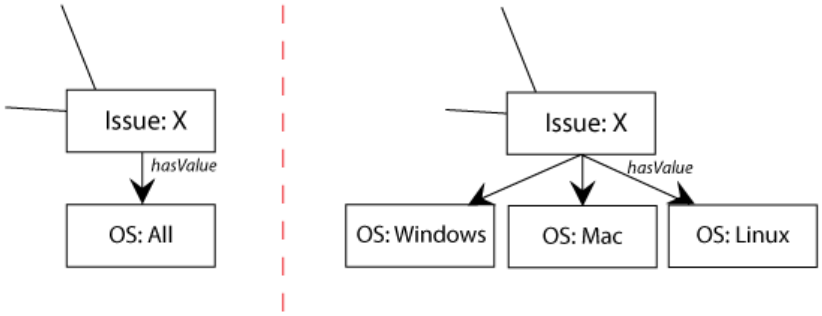

Fig. 11.4. Modelling of bug tracker properties

The "component" and "sub-component" properties of bugs represent another challenge in modelling bug trackers, which is an incomplete representation of the modelled data. The two properties represent a part-of relationship that in reality is recursive (a component can be part of another component, which then again can be part of a component and so on). In bug trackers, only two levels of this recursive relationship are represented. As mentioned above, these fields are also treated as text and obvious dependencies between them are omitted. A bug tracker ontology can use transitivity to automatically allow to resolve this part-of relationship.

Other interesting property values that can be more accurately mapped using ontologies are the "unknown" and "all" fields, which are found in the platform or component property of an issue. Bug trackers treat such fields as simple text, without further semantic meaning and therefore incorrectly answer queries about those fields (see Figure 11.4). For example, a query about all bugs occurring on Windows will not return bugs that have the value "all" set. Similarly, the "unknown" value can be modelled much more accurately using the open world assumption of ontologies. Many classical domain driven ontologies model these property values the same way they occur in the bug trackers itself, i.e., as textual values, and therefore miss out on some of the hidden semantics.

There often exist further properties that are specific to the modelled issue tracker, but they are not further discussed here. While it is the goal of this chapter to describe a general bug tracker ontology, specialized sub-ontologies

might be added to model such properties and relations. The ontology design is derived from several existing software engineering ontologies such as [47] and [30]. However, a common shortcoming among these existing ontologies is their focus on pure conceptualization. Their ontology design lacks necessary concepts, relations, and properties to be able to take advantage of reasoning services.

11.6 Quality classification

Many large software projects face the challenge of managing and evaluating a considerable number of bug reports. This is the case in particular for open source projects, where users can freely (after having registered) submit their own bug reports. For our case study, we selected ArgoUML, a leading UML editor with a publicly accessible bug tracking system. ArgoUML has since its inception in 1998 undergone several release cycles and is still under active development. Its bug database counts over 5,100 open/closed defects and enhancements. In what follows, we describe the data set extracted from the ArgoUML bug repository and the NLP techniques used to mine the bug descriptions. At the end of the section, we provide a discussion on the observed results from our automated analysis of the bug description quality in ArgoUML.

The ontology export support provided by the *General Architecture for Text Engineering (GATE)* [14] framework can be used to integrate knowledge about the quality of bug reports into the existing bug tracker ontology. The newly enriched bug tracking sub-ontology becomes an integrated part of the already existing software engineering ontology. In what follows, we explain in more detail the extraction method applied for each quality attribute. The attributes themselves are derived from results observed in [9] [10] and general guidelines for good report qualities. We define the quality attributes and illustrate them through bug excerpts extracted from the ArgoUML bug repository. Keywords and key expressions are highlighted in bold. As part of the GATE framework, we make use of sentence splitters (identification of sentences), part-of-speech taggers (identification and usage of nouns, verbs, etc.), gazetteer lists (annotation of specific words or sentences) and *Java Annotation Patterns Engine (JAPE)* grammars (annotation of more complex patterns).

Certainty. The level of speculation is embedded in a bug description. A high certainty indicates a clear understanding of the problem and often also implies that the reporter can provide suggestions on how to solve it.

> "Individual parts won't link after downloading I am new to Java, hence this is **probably** a very simple error and **not a 'true' bug**. When I type..." *(Bug Nr.333)*

In [31], it has been demonstrated that hedges can be found with high accuracy using syntactic patterns and a simple weighting scheme. The gazetteer

lists used have been provided by the authors and are used in our approach to identify speculative language. Due to the availability of a negation-identifier, it was further possible to add additional hedging cues based on negated verbs and adjectives (e.g., "not sure"). As suggestions to solve a problem also make use of hedging, a distinction between problem description and suggested solution has to be made. Since problem descriptions tend to appear at the start of a bug report, while suggestions tend to appear at the end, only hedges found in the first half of an error report have been counted. Additionally, the default GATE sentence splitter has been modified to correctly tag question-sentences.

Focus. The bug description does not contain any off-topic discussions, complaints, or personal statements. Only one bug is described per report.

> "V0.10 on OS X has no menu bar When launching v0.10 on OSX, no menu bar is visible. **Additionally**, none of the hot keys work (like Ctrl-S for save)..." *(Bug Nr.860)*

The focus of bug reports is assessed by identifying emotional statements (such as "love" or "exciting"), as well as topic splitting breaks (such as "by the way" or "on top of that") through a gazetteer.

Reproducibility. The bug report description includes steps to reproduce a bug or the context under which a problem occurred.

> "Checking if names are unique **First**, create two packages and one class diagram by package. **Then**, add one class to a package..." *(Bug Nr.79)*

By manually evaluating over 500 bug reports, time clauses used in bug descriptions could be identified as a reliable hint for paragraphs describing the context in which a problem occurred. For example, "When I clicked the button" or "While starting the application". These can easily be annotated using a part-of-speech tagger and JAPE grammar. To identify the listing of reproduction steps, the standard GATE sentence splitter has been modified to recognize itemizations (characters '+', '-', '*') as well as enumerations (in the form of '1.', '(1)', '[1]').

Observability. The bug report contains a clear observed (positive or negative) behaviour. Evidence of the occurred problem such as screenshots, stack traces, or code samples is provided.

> "The GUI **hangs** (CPU load for the Java process jumps to 90 + and does not stop) when I try to change the style of a text object..." *(Bug Nr.364)*

To identify observations in bug descriptions, word frequencies have been compared with the expected numbers from non-bug related sources. For words appearing distinctively more often than expected, a categorization in positive and negative sentiment has been performed.

11.7 Classification

In order to build a solid classification for the above measurements, a random data sample consisting of 178 bugs from all available bug reports in the ArgoUML bug repository was collected. Seven experienced Java developers (master and Ph.D. students which have previously worked with ArgoUML at the source code level) were asked to fill out a questionnaire assessing the quality of bugs. For each of the selected bugs, the users performed an evaluation of the bug report quality using a scale ranging from 1 to 5 (with 1 corresponding to very high quality and 5 to very low quality). The evaluation was performed within one week as part of an assignment.

Predicted	**Observed**					
	Very good	Good	Average	Poor	Very poor	***Precision***
Very good	23	17	9	0	0	*81%*
Good	18	26	3	4	1	*90%*
Average	5	3	3	1	1	*53%*
Poor	0	4	2	3	3	*66%*
Very poor	0	4	2	1	5	*50%*
Recall	*89%*	*85%*	*42%*	*55%*	*80%*	

Fig. 11.5. Decision tree classification of bugs

Figure 11.5 shows the decision tree model with precision and recall for the different quality assessments. The columns denote the average quality rating observed by developers. Rows show the quality predicted by our approach. Dark grey cells show a direct overlapping between the predicted quality and the one rated by developers. In addition, the light grey areas include the predictions which have been off by one from developer ratings. As expected, the classification of bugs with good quality tends to be easier than identifying poor quality bugs. Out of the decision tree model, the following definitions have been generated to perform a classification in the ontology:

```
GoodQualityIssue:
 (hasQuality has VeryGood_Certainty)
   or (hasQuality has Good_Certainty)
 (hasQuality has VeryGood_Focus)
   or (hasQuality has Good_Focus)
```

```
 (hasQuality has VeryGood_Observability)
   or (hasQuality has Good_Observability)
   or (hasQuality has Average_Observability)
 (hasQuality has VeryGood_Reproducibility)
   or (hasQuality has Good_Reproducibility)
 (hasQuality some VeryGood)

AverageQualityIssue:
 (hasQuality has Average_Certainty)
   or (hasQuality has Poor_Certainty)
   or (hasQuality has Good_Certainty)
 (hasQuality has Average_Observability)
   or (hasQuality has Good_Observability)
 (hasQuality has Average_Reproducibility)

PoorQualityIssue:
 (hasQuality has VeryPoor_Certainty)
   or (hasQuality has Poor_Certainty)
   or (hasQuality has Average_Certainty)
 (hasQuality has VeryPoor_Reproducibility)
  or (hasQuality has Poor_Reproducibility)
 (hasQuality has VeryPoor_Observability)
```

11.8 Traceability and querying

Given the common ontological representation, knowledge about the quality of bug reports can be used to guide the knowledge exploration process to support various software evolution activities. Additionally, ontologies can be extended with new concepts or relations to reflect newly gained knowledge. Furthermore, relations among artefact boundaries are possible and can be defined and artefacts can be connected through common (shared) concepts. For example, in the bug tracker and revision control ontologies, bug comments often contain information that refers to revisions, while commit messages mention specific bugs that have been solved within a particular commit. By aligning the two ontologies, either manually or by using text mining techniques, it is possible to answer the following queries:

1. Which files have been changed while working on an issue?
2. Who has been the main developer working on an issue?
3. How much time has been spent resolving an issue?
4. In which version has the issue been resolved?

Additionally, information about the developer working on an issue can sometimes be found as a textual annotation attached to an issue within a bug tracker. Nevertheless, it has to be emphasized that this information is neither

guaranteed to be correct, nor complete. By using ontologies as a common representation for all software artefacts, information is uniformly modelled and can be accessed as well as queried. The following queries (based on the SPARQL syntax) illustrate the use of our bug quality assessment in different contexts.

Query 1: Focuses the maintainer's attention on the classes that are mentioned in quality bug reports (good quality). Helps maintainers/manager prioritize bugs based on the quality of their description.

```
PREFIX Issue:<http://aseg.cs.concordia.ca/ontologies/issue#>
SELECT DISTINCT ?bug
WHERE {
    {
    ?bug Issue:hasPriority Issue:Priority_VeryHigh.
    ?bug rdf:type Issue:GoodQualityIssue.
    }
}
```

Query 2: Identifies users who have submitted low quality bug reports. The query can be applied, for example, to provide additional training or guidance to these users on how to write good bug reports.

```
PREFIX Issue:<http://aseg.cs.concordia.ca/ontologies/issue#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?user
WHERE {
    {
    ?user Issue:isReporterOf ?bug.
    ?bug rdf:type Issue:PoorQualityIssue.
    }UNION{
    ?user Issue:isReporterOf ?bug.
    ?bug rdf:type Issue:AverageQualityIssue.
    }
} LIMIT 20
```

Query 3: Lists bugs blocking other high priority bugs. Helps to identify bugs that need to be resolved prior to any other bugs and therefore, help in the bug triage problem.

```
PREFIX Issue:<http://aseg.cs.concordia.ca/ontologies/issue#>
SELECT *
WHERE {
   ?bug Issue:blocks  [
     Issue:hasPriority Issue:Priority_High
   ] .
   OPTIONAL {
```

```
        ?bug Issue:hasAssignee ?dev .
    }
}
```

Query 4: Queries across the issue ontology and an integrated revision control ontology. Looks for a keyword inside the bug tracker summary and list related files that have been committed with the bug. This can help the developer to find files related to a specific keyword (e.g., to narrow down the files to look at in a maintenance request).

```
PREFIX Issue:<http://aseg.cs.concordia.ca/ontologies/issue#>
PREFIX Revision:<http://aseg.cs.concordia.ca/ontologies/revision#>
SELECT DISTINCT ?summary ?file
WHERE {
   ?bug Issue:issueSummary ?summary .
   ?bug Revision:hasFile ?file .
   FILTER (REGEX(?summary,''KEYWORD'')) .
} LIMIT 1000
```

11.9 Discussion

The Semantic Web is characterized by decentralization and heterogeneity. Given such an environment, knowledge integration, as we performed for the software domain, becomes also the management of inconsistent information. It is not realistic to expect that all sources share a single and consistent view at all times. Rather, we expect disagreement between individual users and tools during an analysis. Trustworthiness within our software engineering ontology is managed through queries which can now be extended similarly with quality attributes. For example, choosing between two bug reports describing a certain portion of source code, one as a 'Composite Pattern' and the other as a 'Singleton', can be resolved by trusting the bug report with higher quality.

There exists a significant body of work which has studied bug reports to automatically assign them to developers [12], assign locations to bug reports [11], track features over time [22], recognize bug duplicates [13], and predict effort for bug reports [45]. Antoniol et al. [5] pointed out that there often exists a lack of integration between version archives and bug databases. The Mylyn tool by Kersten and Murphy [28] allows attaching a task context to bug reports, so that they can be traced at a very fine level of detail.

There exists however only limited work on modelling and automatically evaluating the quality of bug reports themselves. The work most closely related to ours is by Bettenburg et al. and their QUZILLA tool [9]. They also evaluate quality of bug reports, using different quality attributes. Our work can be seen as a continuation of the work performed by Bettenburg. Our reproducibility attribute is a refinement of Bettenburg's [10] attribute, by

considering also the context described in the bug report. We extend the observability property also with negative observations to be further analysed. Furthermore, we introduce the certainty and focus property. Certainty evaluates the confidence level of the bug writer in analysing and describing the bug. Our focus property, on the other hand, looks at emotions and other prose text that might bloat the bug description and make it less comprehensible. Ko et al. [33] performed a linguistic analysis of bug reports, but it lacks both a concrete application and an evaluation of their approach. Their work focuses on bug titles, while our work analyses the full bug description.

Ontologies have been commonly regarded as a standard technique for representing domain semantics and resolving semantic ambiguities. Existing research on applying Semantic Web techniques in software maintenance mainly focuses on providing ontological representation for particular software artefacts or supporting specific maintenance tasks [26]. The introduction of an ontological representation for software artefacts allows us to utilize existing techniques such as text mining and information extraction [21] to "understand" parts of the semantics conveyed by these informal information resources and thus, integrate information from different sources at finer granularity levels. Integrating knowledge about internal and external quality aspects of software artefacts is an important step towards providing semantic support in software evolution.

The ontology and queries outlined in this chapter are only example of the usage of Semantic Web technologies in Software Engineering. In the future, more and more tools will provide RDF data by default, eliminating the need to pre-process and extract information from them. The advantages of this open approach to information sharing in which data is uniquely identified (using RDF) and can be easily consumed by any client is becoming clearer for tool producers. Further, specialized ontologies which allow software engineers to make use of this publicized knowledge will become available and integrated into other tools such as development environments.