Semantic web enabled software analysis

Jonas Tappolet[a,1,*], Christoph Kiefer[a], Abraham Bernstein[a]

[a]*Dynamic and Distributed Information Systems, University of Zurich, Switzerland*

Abstract

One of the most important decisions researchers face when analyzing software systems is the choice of a proper data analysis/exchange format. In this paper, we present EvoOnt, a set of software ontologies and data exchange formats based on OWL. EvoOnt models software design, release history information, and bug-tracking meta-data. Since OWL describes the semantics of the data, EvoOnt (1) is easily extendible, (2) can be processed with many existing tools, and (3) allows to derive assertions through its inherent Description Logic reasoning capabilities. The contribution of this paper is that it introduces a novel software evolution ontology that vastly simplifies typical software evolution analysis tasks. In detail, we show the usefulness of EvoOnt by repeating selected software evolution and analysis experiments from the 2004-2007 Mining Software Repositories Workshops (MSR). We demonstrate that if the data used for analysis were available in EvoOnt then the analyses in 75% of the papers at MSR could be reduced to one or at most two simple queries within off-the-shelf SPARQL tools. In addition, we present how the inherent capabilities of the Semantic Web have the potential of enabling new tasks that have not yet been addressed by software evolution researchers, e.g., due to the complexities of the data integration.

Keywords: Software Comprehension Framework, Software Release Similarity, Bug Prediction, Software Evolution

1. Introduction

Ever since software is being developed, there was a need to understand how code works and why developers made certain decisions. One reason is the fluctuation rate in development teams requiring new employees to familiarize with the existing code and its peculiarities. Secondly, many programmers agree that they tend to forget about a certain piece of code's structure and its rationale behind after a short period of not touching that specific component or class. Another obstacle for comprehending source code is outdated comments [1] or the complete lack thereof. The problem gets aggravated when the history of source code is considered as well; since multiple versions add another dimension of complexity.

*Corresponding author. Tel: +41 44 635 71 47

Email addresses: tappolet@ifi.uzh.ch (Jonas Tappolet), christoph.kiefer@gmail.com (Christoph Kiefer), bernstein@ifi.uzh.ch (Abraham Bernstein)

[1]Partial support provided by Swiss National Science Foundation award number 200021-112330

Imagine a software developer who newly joins a software engineering team in a company: He will most probably be overwhelmed by the vast amount of source code, versions, releases, and bug reports floating around. In the last decades a reasonable amount of different code comprehension frameworks were proposed and implemented. These frameworks aim at facilitating the navigation through the code and the identification of certain anomalies (e.g. code smells [2], anti-patterns [3]) or the structure of the code in general (e.g. intensity of couplings between components). Most of these frameworks first convert the source code into an internal representation that serves as a basis for fast query answering. We will review a selection of such frameworks in Section 2.

In addition, due to the complexity of software products and the growing popularity of open source software components, modern software has become a fine-grained composition of a multitude of different libraries. A typical software product uses external libraries, for instance, for the user interface (e.g. SWT[2]), data layer abstraction (e.g. Hibernate[3]) or logging (e.g. log4j[4]). Each of those libraries in turn make use of sub-libraries that are again maintained by their own project teams. This view turns a software project that seems to be developed locally into a node in a world-embracing network of interlinked software source code or, more technically speaking, a global call-graph. Problems in software projects often happen at this edge between the project's source code and an imported library (as witnessed by the special bug category "3rd party" in bug-trackers): A bug in a library may influence the behavior of the calling component or the wrong usage of a component may lead to instabilities in the code. Therefore, this global cloud of software source code and its related information (versions, releases and bug reports) implies additional requirements for a comprehension framework. Instead of an insular system with internal representations, each software project participating in the cloud needs to exhibit its information in an open, accessible and uniquely identifiable way. To this end, we propose the usage of semantic technologies such as OWL, RDF and SPARQL as a software comprehension framework with the abilities to be interlinked with other projects. We introduce a set of ontologies and techniques to represent software project information semantically and focus on the general abilities of semantic technologies to cover everyday problems in software comprehension (also referred to as Software Analysis). We show that semantic technologies indeed bear the potential to serve as a general-purpose framework and we believe that implicit abilities such as the strong web-based foundation including unique resource identifiers or distributed querying are the key towards a worldwide connection of different software projects.

Specifically, we present our software evolution ontology EvoOnt, which in fact is a graph-based, self-describing representation for source code and software process data, allows the convenient integration, querying, and reasoning of the software knowledge base. Together with some standard Semantic Web tools and our domain-independent iSPARQL as well as SPARQL-ML query engines, EvoOnt can help to resolve various software analysis tasks (including some in cross-project settings).

[2] `http://www.eclipse.org/swt/`

[3] `http://www.hibernate.org/`

[4] `http://logging.apache.org/log4j/index.html`

EvoOnt is a set of software ontologies based on OWL. It provides the means to store all elements necessary for software analyses including the software design itself as well as its release and bug-tracking information. Given the proliferation of OWL (as a standard), a myriad of tools allow its immediate processing in terms of visualization, editing, querying, reasoning, and debugging avoiding the need to write code or use complicated command line tools. OWL enables handling of the data based on its semantics, which allows the simple extension of the data model while maintaining the functionality of existing tools.

Furthermore, given OWLs Description Logic foundation, any Semantic Web engine allows deriving additional assertions in the code such as orphan methods (see Section 5.4), which are entailed from base facts. To highlight EvoOnts full capabilities we used it in conjunction with two of our domain-independent SPARQL extensions iSPARQL [4] and SPARQL-ML [5]. iSPARQL extends the Semantic Web query language SPARQL with similarity joins allowing EvoOnt users to, e.g., query for similar software entities (classes, methods, fields, etc.) in an EvoOnt dataset or to compute statistical propositions about the evolution of software projects for instance (see Section 5.2). SPARQL-ML seamlessly extends SPARQL with (two) machine learning libraries allowing, e.g., to use SPARQL queries for induction of defect prediction models.

The main contribution of this paper is the introduction of EvoOnt simplifying most typical software analysis and prediction tasks within one extendible framework. In addition, the simplicity in which EvoOnt could support the software analysis indicates that future tasks might be just as easy to handle. Last but not least, the inherent capability of the Semantic Web to process distributed knowledge bases significantly simplifies analyses among many different software projects. We show that our approach allows reducing more than 75% of the evolution analysis tasks typically conducted at the ICSE Mining Software Repository Workshop (MSR) to one (sometimes two) queries and argue that some other tasks could also be performed with some simple extensions of EvoOnt and/or iSPARQL/SPARQL-ML.

The remainder of this paper is structured as follows: next, we succinctly summarize the most important related work. Section 3 presents EvoOnt itself, which is followed by brief introductions to iSPARQL and SPARQL-ML. Section 5 illustrates the simplicity of using EvoOnt. To close the paper, Section 6 presents our conclusions, the limitations of our approach, and some insight into future work. We would like to mention that this work builds upon two of our previous publications [6, 7].

2. Related work

2.1. Software comprehension frameworks

As mentioned in the section above, a number of software comprehension frameworks have been proposed in recent research. We will give a brief overview of these frameworks with a focus on the applicability to a worldwide-interweaved scenario. One representative dating from the late 80ies is RIGI [8]. It has a strong emphasis on the recovery of the code's architectural structure. To that end, RIGI uses an internal representation that is a graph, however, mapped to a relational data model for storage. RIGI comes with a number of different components such as a GUI and analysis component. While RIGI has a strong

focus on visual analysis, it is limited to the implemented analysis methods because the internal representation is not exposed and accessible for third-party tools. The same applies to tools like CIA [9] and CShape that were implemented with a finite set of analysis tasks in mind. This is a gap that is filled by the GENOA [10] framework. It also has a graph-based internal representation, but additionally offers a formal language interface that can be used by a multitude of different frontends (analysis tools) to retrieve information. GUPRO [11] follows a similar approach by using a graph-based query language to access the source code information. LaSSIE [12], presented by the same authors as GENOA, exposes a natural language interface serving as a more intuitive way of accessing the knowledge base. All of the above mentioned approaches use internal data models with local identifiers. It is non-trivial to put those software projects into relation to their libraries and dependencies. They provide non-standardized query interfaces to access their knowledge base, if at all. Finally, the OMG (Object Management Group) specified QVT [13], for instance used by the ADM initiative (Architecture-Driven Modernization). QVT stands for *Query / View / Transformation* and its goal is the transformation between different object-oriented code model representations. Unlike SPARQL, QVT uses mostly an SQL-like relational approach instead of graph patterns. The tool support is not (yet) very comprehensive.

2.2. Software exchange formats

To address the issue that each analysis framework needs to provide its own extraction tools suitable for the internal format, generic exchange formats have been proposed. Many of the above-mentioned tools define their own format primarily for persistent storage of their internal data. With GXL [14], an effort was made to exchange software graphs between TA [15], TGraphs (GUPRO), RPA [16], RSF (Rigi Standard Format) [8] and PROGRES [17]. It extends the tree-based XML to be able to express graphs. An earlier exchange format was CDIF (CASE Data Interchange Format), an EIA[5] standard for exchanging data between CASE (Computer Aided Software Engineering) tools. It uses flat textual representations, which makes it human-readable. An example for a comprehension framework supporting CDIF is FAMIX, the meta-model for object-oriented source code of the MOOSE[6] project; in the earlier versions of FAMIX the CDIF format was used. Later, the successor, XMI (XML Metadata Interchange) [18], an XML based exchange format able to express multiple different models and even graphics was used. XMI is a standard of the OMG. Both CDIF and XMI are highly sophisticated exchange formats. Since they were designed especially for the domain of CASE tools there is a good and widespread tool support. Unfortunately, tool providers tend to extend XMI with proprietary elements resulting in an erosion of the standard. Another downside is the need for transformation between a tool's internal representation and the exchange format. This can be an error-prone and expensive step. Our approach proposes the usage of one format both for internal representation and as exchange format. In addition, neither XML nor CDIF impose the rigid usage of global identifiers in a way that RDF does. This is, as mentioned above, a precondition for inter-project software

[5] `http://www.eia.org/`

[6] `http://moose.unibe.ch/`

comprehension and code analysis. Finally, none of the existing exchange formats expose their semantics formally. They are usually defined in a human-readable format aiming at being implemented in tools. The advantage of self-describing and exposed semantics is the fact that tools can handle the information without the need of being developed for a certain domain of application (e.g. query languages, visualization tools or machine learning tools).

2.3. Semantic web enabled software engineering

Semantic Web technologies have successfully been used in recent software engineering research. For example Dietrich [19] proposed an OWL ontology to model the domain of software design patterns [20] to automatically generate documentation about the patterns used in a software system. With the help of this ontology, the presented pattern scanner inspects the abstract syntax trees (AST) of source code fragments to identify the patterns used in the code.
The decision as to which software design patterns to choose is a crucial step in designing a software system. Choosing a wrong (or inappropriate) architectural design probably results in high maintenance costs and poor performance and scalability. With the proposed software evolution ontology EvoOnt we are, in fact, able to measure the quality of software in terms of its used design patterns. This, in combination with data from version control and a bug-tracking system, enables us to perform powerful and complex software analysis tasks (see Section 5).

Highly related is the work of Hyland-Wood [21], in which the authors present an OWL ontology of *Software Engineering Concepts (SECs)*. Using SEC, it is possible to enable language-neutral, relational navigation of software systems to facilitate software understanding and maintenance. The structure of SEC is very similar to the language structure of Java and includes information about classes and methods, test cases, metrics, and requirements of software systems. Information from versioning and bug-tracking systems is, however, not modeled in SEC.
In contrast to EvoOnt, SEC is not based on FAMIX [22] that is a programming language-independent model to represent object-oriented software source code. EvoOnt is, thus, able to represent software projects written in many different object-oriented programming languages.

Witte *et al.* [23] presented an approach that is similar to the idea of EvoOnt. The scope of their work is not the integration of bug, version and source code information but the connection of source code with its documentation. We believe that EvoOnt could be attached to the documentation ontologies of their work to have even more information available in the knowledge base.

Both, Mäntylä [24] and Shatnawi [25] carried out an investigation of *code smells* [2] in object-oriented software source code. While the study of Mäntylä additionally presented a taxonomy (*i.e.*, an ontology) of smells and examined its correlations, both studies provided empirical evidence that some code smells can be linked with errors in software design.

Happel [26] presented the *KOntoR* approach that aims at storing and querying metadata about software artifacts in a central repository to foster their reuse. Furthermore, various

ontologies for the description of background knowledge about the artifacts such as the programming language and licensing models are presented. Also, their work includes a number of SPARQL queries a developer can execute to retrieve particular software fragments which fit a specific application development need.

Finally, we would like to point out that EvoOnt shares a lot of commonalities with Baetle[7] which is an ontology that heavily focuses on the information kept in bug databases, and makes use of many other well-established Semantic Web ontologies, such as the Dublin Core[8], and FOAF[9]. We merged the ideas realized in Baetle with our Bug Ontology Model (see Section 3.3). Therefore, most members of the Baetle community base their work on our ontology.

As a conclusion of the related work, we believe that EvoOnt will contribute to the state of the art as follows: (1) The usage of the open and well-established RDF/OWL format can decouple the analysis tool from the data export tool (so far, an analysis tool is responsible to transform the data into its own internal format). (2) Other than existing exchange formats, EvoOnt exposes its semantics, which allows standard tools to process the data using a unified query language (SPARQL) including extensions such as iSPARQL and SPARQL-ML. Additionally, unlike CDIF or XMI, EvoOnt can be extended easily by either attaching additional ontologies or by using sub-concept specialisation. Finally (3), EvoOnt imposes the usage of globally unique identifiers, which is a main requirement for inter-project analysis.

3. Software ontology models

In this section, we describe our OWL software ontology models. Figure 1 shows the complete set of our ontologies and their connections between each other. We created three different models which encapsulate different aspects of object-oriented software source code: the *software ontology model (som)*, the *bug ontology model (bom)*, and the *version ontology model (vom)*. These models not only reflect the design and architecture of software, but also capture information gathered over time (*i.e.*, during the whole life cycle of the project). Such meta-data includes information about revisions, releases, and bug reports. We connected our ontologies to existing ones from other domains. A bug report for example can be seen as a representation of a work flow. Therefore, we used the defined concepts of Tim Berners-Lee's work flow ontology[10]. The following list shows the external ontologies with their description and abbreviation (prefix) used in the remaining parts of this paper.

- *doap: Description of a Project* defining concepts about a project itself as well as different version control systems (e.g. `CVSRepository`)
- *sioc: Semantically Interlinked Online Communities.* In this ontology, concepts modeling the activities of online communication are defined.

[7] `http://code.google.com/p/baetle/`
[8] `http://dublincore.org/documents/dcq-rdf-xml/`
[9] `http://www.foaf-project.org/`
[10] `http://www.w3.org/2005/01/wf/`

- *foaf:* The *Friend Of A Friend Ontology* is an approach of modelling social networks, i.e., persons and the connection to each other. We use the concept `Person` to reflect human interaction within the repositories.
- *wf:* Tim Berners-Lee's *work flow ontology*. In our approach, a bug report is considered a work flow. Therefore, an issue (bug report) is a `wf:Task` which can have a `wf:NonTerminalState` (still processing) or a `wf:TerminalState` (fixed / closed bug). This idea is adopted from the *baetle* project.

3.1. Software ontology model

Our software ontology model (som) is based on *FAMIX* (FAMOOS Information Exchange Model) [22], a programming language-independent model for representing object-oriented source code. FAMIX and other meta-models abstract OO concepts in a similar way. Therefore, the choice of using FAMIX is not irrevocable. Other meta-models such as Lethbridge *et al.*'s DMM (Dagstuhl Middle Metamodel)[27] can partially (or completely) be attached to the current ontology to, for example, express control structures such as `while`, `switch`, or `if`. This can be achieved by using `sameAs` relations or, in case of a finer-grained definition, a subclass definition — another advantage of RDF/OWL's exposed semantics.

On the top level, the ontology specifies `Entity` that is the common superclass of all other entities, such as `BehaviouralEntity` and `StructuralEntity` (see Figure 1 (top)). A `BehaviouralEntity` represents the definition of a behavioural abstraction in source code, *i.e.*, an abstraction that denotes an action rather than a part of the state (achieved by a method or function). A `StructuralEntity`, in contrast, represents the definition in source code of a structural entity, *i.e.*, it denotes an aspect of the state of a system [22] (*e.g.*, variable or parameter).

When designing our OWL ontology, we made some changes to the original FAMIX: we introduced the two new classes `Context` and `Namespace`, the first one being the superclass of the latter one. `Context` is a *container class* to model the context in which a source code entity appears. `Namespace` (not to confuse with an RDF namespace, *i.e.* URI) denotes a hierarchical identifier for source code (*e.g.*, in Java this concept is called *package*). Taking advantage of RDF's graph-based foundation, RDF/OWL now allows us to elegantly model so-called *association classes*, such as methods accessing a variable with the property `accesses` having the domain `BehaviouralEntity` and range `StructuralEntity`.

3.2. Version ontology model

The goal of our version ontology model (vom) is to specify the relations between files, releases, and revisions of software projects and the projects themselves (See Figure 1 (middle)). We took the data model of Subversion[11] as a blueprint for vom. To that end, we defined the three OWL classes `Path`, `Release`, and `Version` as well as the necessary properties to link these classes. A `Path` denotes a constant, non-temporal entity which could also be seen as

[11] `http://subversion.tigris.org/`

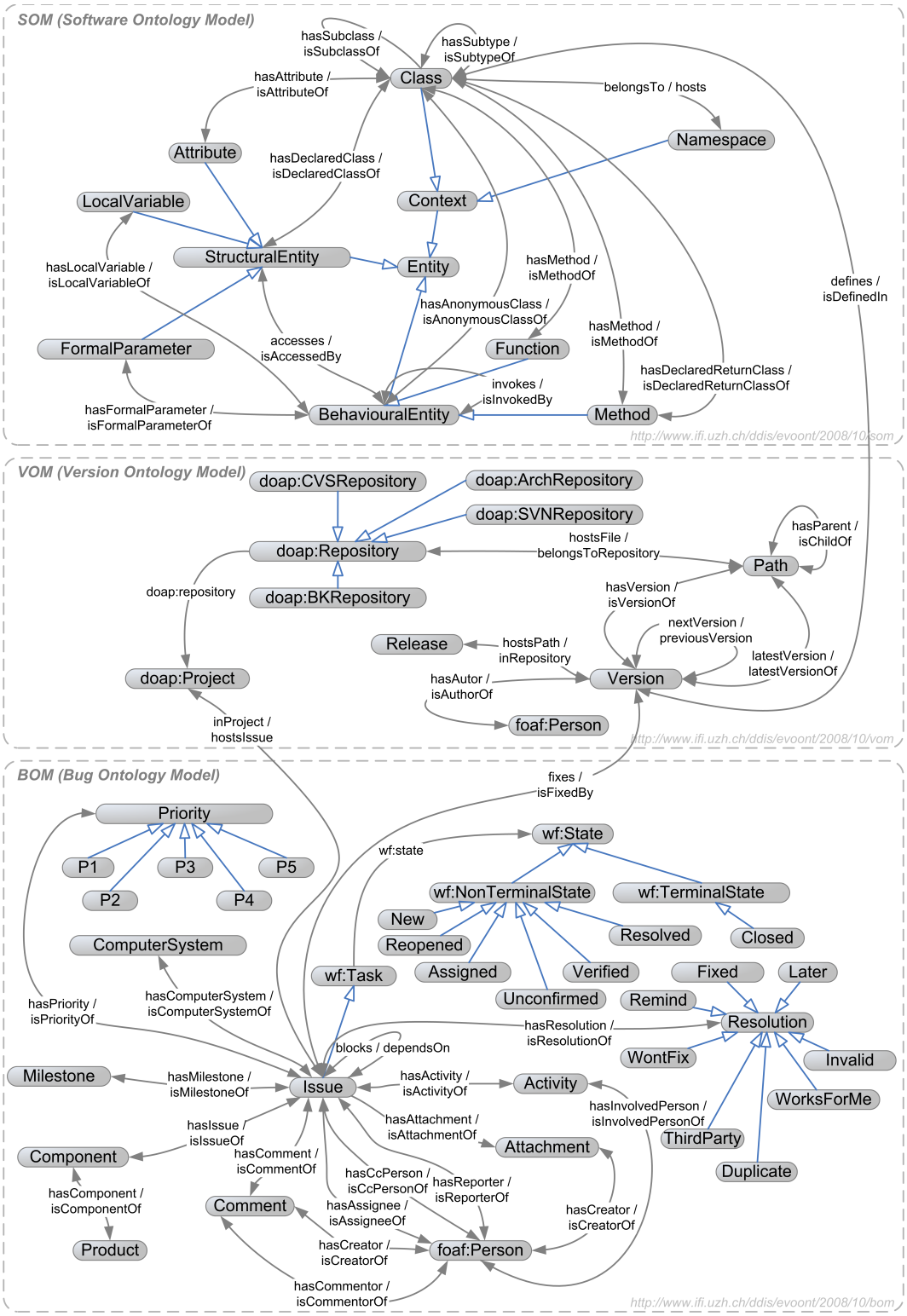


Figure 1: EvoOnt's three ontology models: Software (SOM), Version (VOM), and Bug (BOM) Ontology Model. Solid arrows: property relationships; hollow arrows: class inheritance.

a file, but without content or any meta-data. According to the name, a `Version` relates to a file belonging to a `Path` which is valid for a certain timespan between its predecessor and successor version. A `Version` has content and meta-data as we would expect it from a classical file in a file system (i.e., author or creation date) and, as a characteristic to a versioning system, a comment and a revision number. As container entity we import `doap:Project` and `doap:Repository` (with its subclasses) from the *Description Of A Project* ontology. Every path is connected to `Repository` using the `belongsToRepository` object property. On the other hand, a `Repository` is connected to a `Project` that it is part of using the `doap:repository` property. A `Path`, for example, has a number of revisions and, therefore, is connected to `Revision` by the `hasRevision` property. At some point in time, the developers of a software project usually decide to publish a new release, which includes all the revisions made until that point. In our model, this is reflected by the `isReleaseOf` property that relates `Release` with one or more `Version`s.

3.3. Bug ontology model

Our bug ontology model (bom) (see Figure 1 (bottom)) is inspired by the bug-tracking system *Bugzilla*.[12] `Issue` is the main class for specifying bug reports. As stated above, we consider a bug report to be a task from a work flow. Therefore, `Issue` is a subclass of `wf:Task` defined in the workflow ontology. It is connected to a `foaf:Person`, which stands for any person interacting with the bug tracking system, *i.e.*, the reporter of a bug or a developer that fixes a bug. `Activity` stands for every change made to a bug report. This can be, *e.g.*, the change of the status or resolution of a bug report or the current state of the bug.[13] `Issue` has a connection to `Version` (see Section 3.2) via the `isFixedBy` property. This way, information about which file version successfully resolved a particular bug can be modeled, and vice versa, which bug reports were issued for a specific source code file.

4. Semantic web query methods for software analysis

The contribution of our paper is to show how software analysis tasks can be vastly simplified using EvoOnt and off-the-shelf Semantic Web tools. To ensure that this paper is self-contained we succinctly review the two non-standard, off-the-shelf, domain-independent Semantic Web query approaches used in this paper: iSPARQL (imprecise SPARQL) and SPARQL-ML (SPARQL Machine Learning). For details about these approaches, refer to [28], [4], and [5] respectively.

4.1. iSPARQL

iSPARQL[14] is an extension of SPARQL [28]. It introduces the idea of *virtual triples*—triples that are not matched against the underlying ontology graph, but used to configure

[12] `http://www.bugzilla.org/`

[13] `https://bugs.eclipse.org/bugs` shows various concrete examples.

[14] A demonstration of iSPARQL is available at `http://www.ifi.uzh.ch/ddis/isparql.html`.

similarity joins. Similarity joins specify which pair(s) of variables (that are bound to resources in SPARQL) should be joined and compared using a certain type of similarity measure. Thus, they establish a *virtual relation* between the resources. A similarity ontology defines the admissible virtual triples and links the different measures to their actual implementation in SimPack—our library of similarity measures.[15] For convenience we list some similarity measures used by iSPARQL in Table 1. The similarity ontology also enables the specification of more complicated combinations of similarity measures.

Measure	**Explanation**
Levenshtein measure (simple)	String similarity between, for instance, class/method names: *Levenshtein* string edit distance measuring how two strings relate in terms of the number of insert, remove, and replacement operations to transform one string into the other [29].
TreeEditDistance measure (simple)	Tree similarity between tree representations of classes: measuring the number of steps it takes to transform one tree into another tree by applying a set of elementary edit operations: insertion, substitution, and deletion of nodes [30].
Graph measure (simple)	Graph similarity between graph representations of classes: the measure aims at finding the maximum common subgraph (MCS) of two input graphs [31]. Based on the MCS the similarity between both input graphs is calculated.
CustomClassMeasure (engineered)	User-defined Java class similarity measure: determines the affinity of classes by comparing their sets of method/attribute names. The names are compared by the Levenshtein string similarity measure. Individual similarity scores are weighted and accumulated to an overall similarity value.

Table 1: Selection of four iSPARQL similarity strategies.

4.2. SPARQL-ML

Specifically for our bug prediction experiments in Section 5.6, we will use our SPARQL-ML (SPARQL Machine Learning) approach – an extension of SPARQL with knowledge discovery capabilities. SPARQL-ML is a tool for efficient, relational data mining on Semantic Web data.[16] Its syntax and semantics were thoroughly examined in [5] together with a number of case studies to show the usability of SPARQL-ML. In this section we will give a brief introduction and example queries.

SPARQL-ML enables the usage of *Statistical Relational Learning (SRL)* methods such as *Relational Probability Trees (RPTs)* [32] and *Relational Bayesian Classifiers (RBCs)* [33] that take the relations between RDF resources into account for the induction of a model, as well as for making predictions. These methods have been shown to be very powerful for SRL as they model not only the intrinsic attributes of resources, but also the extrinsic relations to other resources [34], and thus, should perform at least as accurate as traditional, propositional learning techniques.

Listing 1 is the SPARQL-ML query which builds up a prediction model (indicated by the `CREATE MINING MODEL` statement). Within this block, the target variables, prediction type and feature types are defined (lines 2–5). The following block (lines 8–10) are triple patterns to bind the variables that serve as features for the prediction in the training period. Finally, in line 12, the library providing the machine learning algorithms is selected (in this case: proximity, weka would be available as well).

[15] `http://www.ifi.uzh.ch/ddis/simpack.html`

[16] SPARQL-ML is available at `http://www.ifi.uzh.ch/ddis/sparql-ml.html`

```
1  CREATE MINING MODEL <http://www.example.org/bugs>
2    { ?file                   RESOURCE TARGET
3      ?error                  DISCRETE PREDICT {'YES','NO'}
4      ?reportedIssues3Months  CONTINUOUS
5      ?reportedIssues5Months  CONTINUOUS
6    }
7  WHERE
8    { ?file rcs:hasError              ?error .
9      ?file rcs:reportedIssues3Months ?reportedIssues3Months .
10     ?file rcs:reportedIssues5Months ?reportedIssues5Months
11   }
12 USING <http://kdl.cs.umass.edu/proximity/rpt>
```

Listing 1: SPARQL-ML induce statement.

```
1  SELECT DISTINCT ?file ?error ?rpt ?prob
2  WHERE
3    { ?file rcs:hasError              ?error .
4      ?file rcs:reportedIssues3Months ?reportedIssues3Months .
5      ?file rcs:reportedIssues5Months ?reportedIssues5Months .
6
7      ( ?rpt ?prob ) sml:predict
8                       ( <http://www.example.org/bugs>
9                         ?file, ?error, ?reportedIssues3Months,
10                        ?reportedIssues5Months )
11   }
```

Listing 2: SPARQL-ML predict statement.

Listing 2 applies the prediction model that was learned in the query of 1 to a test set that is bound with the triple patterns in lines 3-5. Since the learning of a prediction model and its application to the test set are two detached queries, the learned model is passed between those two queries using a URI (line 1 in Listing 1 and line 8 in Listing 2).

5. Experimental evaluation

To show the applicability and ease of use of our approach for a very broad range of Software Analysis tasks we first surveyed the last four years of the proceedings of the *ICSE Workshop on Mining Software Repositories (MSR)*[17] and then tried to replicate as many experiment types as possible with EvoOnt and the off-the-shelf query tools.

The most actively investigated software analysis tasks are shown in Table 2. The table shows the 12 task categories we identified together with their percentage of numbers of papers. Note that these categories were subjectively constructed. We found this procedure very useful to get an overview of current research activities, for which our Semantic Web tools could make a significant contribution. Furthermore, Table 2 also shows for which tasks we have successfully applied one or more of our tools.

[17] `http://www.msrconf.org/`

[18] These tasks could theoretically be accmplished with SPARQL. However, we did not conduct any social network analysis experiments due to the lack of datasets.

Task	Fraction (%)	Domain Independent Tool
General Framework (*e.g.*, facilitate analysis process, data cleansing & integration, repository query language)	13.59	RDF, OWL, SPARQL
Bug/Change Prediction (*e.g.*, build defect detectors/classifiers, bug risk & fixing time prediction)	13.59	SPARQL-ML
Social Network Analysis (*e.g.*, mailing list analysis, understand developer roles & networks, discover development processes)	11.65	—[18]
Software Evolution Analysis (*e.g.*, study & characterize system evolution, visualization)	10.68	iSPARQL
Software Reuse (*e.g.*, code suggestion, similarity analysis, code search & retrieval, clone detection)	10.68	SPARQL, iSPARQL
Mining CVS (*e.g.*, mine local histories)	9.71	SPARQL
Change Impact Analysis (*e.g.*, detect incomplete refactorings, signature change analysis, code smells)	9.71	SPARQL
General Mining (*e.g.*, find sequences of changed files)	8.74	SPARQL
Text Mining (*e.g.*, free text search, mining code comments, keyword search)	4.85	SPARQL, iSPARQL
Source Code Metrics (*e.g.*, code clone coverage)	2.91	SPARQL
Repository Mining Tools (*e.g.*, evaluation of tools)	1.94	—
Pattern Detection (*e.g.*, detect software design patterns, find system-user interaction patterns)	1.94	SPARQL
	100% (103 papers)	

Table 2: Popular software analysis tasks from MSR 2004–2007

Of the accepted 103 papers in total (not including MSR challenge reports), almost 14% are dealing with the construction and evaluation of *General Frameworks* for the integration, cleansing, analysis, and querying of data from various software-development related sources, such as versioning and bug-tracking systems, source code, forums, mailing lists, etc. Our EvoOnt approach *is*, in fact, a unified, general purpose framework integrating software data from diverse sources and enabling its efficient querying and analysis along a multitude of dimensions.

Approximately the same number of papers investigate the task of *Bug and Change Prediction* to find the locations in software that most likely will have to be fixed in the future based on historical information. This is a perfect candidate for our SPARQL-ML tool as it allows us to make a statistical statement about the likelihood of the occurrence of bugs and changes in source code (see Section 5.6).

Another set of 12 papers examines methods from *Social Network Analysis* to, for instance, determine developer roles and to reveal software development processes. We did not yet address any of these tasks with one of our tools. This is not a limitation of our approach and the used techniques themselves but of the data sets available to us. We believe that our tools could be applied to these tasks with comparable performance.

Software Evolution Analysis and *Software Reuse* are the fourth and fifth largest categories. These categories are interesting as they hold tasks such as *evolution visualization*, *similarity analysis*, as well as *code search & retrieval* that can clearly be tackled by our Semantic Web approaches.

Additional categories we found suitable for further consideration are *Change Impact Analysis*, *Source Code Metrics*, and *Pattern Detection*. Specifically, the first one includes *detection of code smells* (*i.e.*, code design flaws) that can partly be solved by approaches falling into the second category to compute *source code metrics*. *Pattern Detection* is in

range of our tools as our FAMIX-based software model approach allows us to query the RDF data set for certain *software design patterns*.

Note that tasks such as *visualization* and *search* are common to almost all categories. We address visualization in Section 5.2, in which we apply iSPARQL to discover and visualize the architectural evolution of software components.

Given these categories, we chose to conduct the following five sets of experiments (in increasing order of complexity):

1. *software evolution measurements*: analyzing and visualizing changes between different releases;
2. *metrics experiments*: evaluation of the ability to calculate object-oriented software metrics;
3. *impact experiments*: evaluation of the applicability of Semantic Web tools to detect code smells;
4. *density measurements* (as a subtask of the evolution and metrics experiments): determining the amount of bug-fixing and "ordinary" software development measured over all software engineering activities;
5. *bug prediction assessments*: showing the usefulness of SPARQL-ML for bug prediction.

5.1. Experimental setup and datasets

For our experiments, we examined 206 releases of the `org.eclipse.compare` plug-in for Eclipse. This plug-in consists in average of about 150 java classes per version. Multiplied with the 206 releases we have the source code information of roughly 30'000 classes in our repository[19]. To generate an OWL data file of a particular release, it was first automatically retrieved from Eclipse's CVS repository and loaded into an in-memory version of our software ontology model, before it was exported to an OWL file. To get the data from CVS and to fill our version ontology model, the contents of the Release History Database (RHDB) [35] for the compare plug-in were loaded into memory and, again, parsed and exported to OWL according to our version ontology model. While parsing the CVS data, the commit message of each revision of a file was inspected and matched against a regular expression to detect referenced bug IDs. If a bug was mentioned in the commit message as, for instance, in *"fixed #67888: [accessibility] Go To Next Difference stops working on reuse of editor"*, the information about the bug was (automatically) retrieved from the web and also stored in memory. Finally, the data of the in-memory bug ontology model was exported to OWL. None of the above steps needed any kind of user interaction (except for selecting the project and versions to export) and were conducted by an Eclipse plug-in allowing us to rely on a multitude of functions provided by the Eclipse framework such as checkout of a release or build-up and traversal of the syntax trees. A general downside of the design of our extraction tool was the generation of an in-memory model before we wrote the data to RDF/OWL.

[19] We believe that lines of code is not a suitable metric in this cases because we use graph-based representations. However, for comparison, the LOC of one version is about 38'000. Multiplied with the 206 versions we have information about approximately 7.8 millions LOC.

This fact limited us in the choice of project sizes because the in-memory models of projects larger than 150 classes per version reached the limit of the physical main memory of the extracting machine. A currently developed 2nd version of the extraction tools now directly generates triples that get immediately written to disk. Therefore, the project size ceases to be a limiting factor.

Recently, Gröner *et al.* [36] compared query approaches in reverse engineering. Specifically, they compared GUPRO/GReQL with OWL/SPARQL and showed that the time costs are, in summary, more than ten times higher for OWL/SPARQL than for GUPRO/GReQL. Note that this performance difference needs to be seen in the light of our other investigations, where we showed that simple selectivity-based query optimization applied to existing SPARQL engines techniques can lead to performance improvements of 3-4 orders of magnitude [37, 38]. In particular, since Gröner *et al.* found that KAON2 was about 1 order of magintude slower than GUPRO/GReQL and we found that our static query optimizer sped up typical SPARQL queries on KAON2 by about 600 times (compared to other SPARQL engines even by about 700 times) [37], we can expect that optimized SPARQL engines should provide an at least equal if not superior performance compared to GUPRO/GReQL. As a consequence, we share Gröner *et al.* 's opinion that recent research [39, 40] will lead to vast improvements in execution-time of SPARQL queries, and, therefore, our approach will most probably have a competitive time complexity in future applications.

5.2. Task 1: software evolution analysis

With the first set of experiments, we wanted to evaluate the applicability of our iSPARQL approach to the task of software evolution visualization (*i.e.*, the graphical visualization of code changes for a certain time span in the life cycle of the Eclipse `compare` software project). This analysis is especially important when trying to detect code clones. To that end, we compared all the Java classes of one major release with all the classes from another major release with different similarity strategies mirroring the experiments of Sager et al. [30] Listing 3 shows the corresponding query for two particular releases and the *Tree Edit Distance* measure. In lines 3–6 and 8–11 of Listing 3, four variables are bound to each class URI and its literal value. Identified by the `IMPRECISE` keyword, each class URI is passed to a property function (`isparql:treeEditDistance`, line 14) which binds the calculated structural similarity to the variable `sim1`. Another similarity algorithm is applied to the class names (`isparql:levenshtein`, line 15). Finally, the two similarities are weighted and combined to an overall score (line 16).

The results of the execution of Listing 3 for the releases 3.1 and 3.2 are shown in Figure 2. The heatmaps mirror the class code changes between the two releases of the project by using different shades of gray for different similarity scores in the interval $[0, 1]$. Analyzing the generated heatmaps, we found that the specialized *Custom Class Measure* performed best for the given task; most likely, this is because it is an algorithm especially tailored to compare source code classes. The combination of method/attribute set comparisons together with the *Levenshtein* string similarity measure for method/attribute names (Figure 2(b)) turned out to be less precise. In all our experiments, the *Graph Measure* (Figure 2(c)) was the least accurate indicator for the similarity of classes. What is common to Figures 2(a–c)

```
1  SELECT ?similarity
2  WHERE
3    { ?class1   som:uniqueName ?name1 ;
4                som:isClassOf  ?file1 .
5      ?file1    som:hasRelease ?release1 .
6      ?release1 vom:name       ''R3_1'' .
7
8      ?class2   som:uniqueName ?name2 ;
9                som:isClassOf  ?file2 .
10     ?file2    som:hasRelease ?release2 .
11     ?release2 vom:name       ''R3_2'' .
12
13     IMPRECISE
14       { ?sim1       isparql:treeEditDistance ( ?class1 ?class2 ) .
15         ?sim2       isparql:levenshtein      ( ?name1 ?name2 ) .
16         ?similarity isparql:score            ( 0.25 ?sim1 0.75 ?sim2 )
17       }
18   }
19 ORDER BY DESC (?similarity)
```

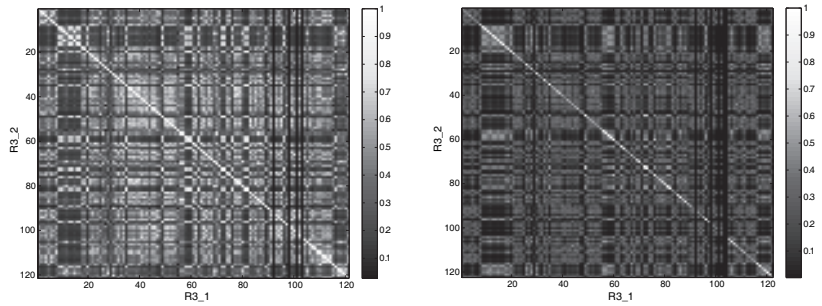
Listing 3: iSPARQL query: Computation of the structural (*Tree Edit Distance*) and textual (*Levenshtein*) similarity between the classes of two releases.

is the diagonal line denoting high similarity of the same classess between different versions. This is an obvious and expected fact because usually only a small percentage of the source code changes between two versions. Another, less obvious fact is the high similarity observed in the top-left area of the figures. This is a cluster of classes very similar to each other, but highly different to the rest of the classes. An in-depth analysis showed that this cluster consists of interface definitions, which lack a lot of features of "normal" classes (e.g. method bodies, variable declarations, anonymous classes). In general, a software project manager or auditor can use the information of these visualizations to get a preselection of possible candidates for duplicate code.

Furthermore, to shed some light on the history of a single Java class, we measured the similarity of the class from one release and the (immediate) next release and repeated this process for all classes and releases. This resulted in an array of values $sim_{class}^{R_i,R_j}$, each value expressing the similarity of the same class of two different releases R_i and R_j. However, to visualize the *amount of change*, we plotted the inverse (*i.e.*, $1 - sim_{class}^{R_i,R_j}$) as illustrated in Figures 2(d–f) that show the history of changes for three distinct classes of the project. There are classes such as `BufferedCanvas` which tend to have fewer changes as the project evolves over time. Other classes such as `CompareEditor` (Figure 2(e)) are altered again and again, probably implying some design flaws or code smells. Then again, there are classes which tend to have more changes over time as shown in Figure 2(f) for the class `Utilities`. This information can also help to manage a software project since it allows managers to allocate resources to classes/components that have a high development activity.

5.3. Task 2: computing source code metrics

With our second set of experiments, we wanted to demonstrate the possibility of calculating *object-oriented software design metrics* [3] using SPARQL. For illustration purposes, we have chosen six of them which we will succinctly discuss in this section. Note that there



(a) CustomClassMeasure (b) TreeEditDistance (c) GraphMeasure

(d) BufferedCanvas (e) CompareEditor (f) Utilities

Figure 2: Figures 2(a–c) depict the computed heatmaps of the between-version comparison of all the classes of releases 3.1 and 3.2 of the `org.eclipse.compare` plugin using three different similarity strategies. Furthermore, the history of changes for three distinct classes of the project is illustrated in Figures 2(d–f).

is a close connection between code smells and software metrics in the sense that metrics are often used to identify possible code smells in object-oriented software systems (see Section 5.4).

5.3.1. Changing methods (CM) and changing classes (CC)

When changing the functionality of a method (*callee*), in most cases this will have an impact to the invoker (*caller*) of this method. Consider a method in an arbitrary class for sorting a list of input strings. For some reason a developer might decide to change the order of the result list, for instance, from ascending to descending order. If the change is not

communicated to the caller, it might be unable to process the results correctly as it expects the returned list to be sorted in ascending order.

This is a typical example for a change that can lead to defects of the invoking methods and classes as they expect different semantics of the callee. Therefore, a method that is invoked by many other methods has a higher risk of causing a defect because a developer might forget to change every invoking method.

With SPARQL we are able to easily count the number of times a method is called. This is shown in Query 4 which lists methods and the number of their distinct callers (variable ?cm; in query). In addition, the query counts the number of distinct classes these calling methods are defined in (variable ?cc; also in query). The topmost answers of the query are shown in Table 3.

```
SELECT ?method (count(distinct ?invoker) AS ?cm)
               (count(distinct ?invokerClass) AS ?cc)
WHERE
  { ?class        som:hasMethod ?method .
    ?invoker      som:invokes   ?method .
    ?invokerClass som:hasMethod ?invoker
  }
GROUP BY ?method
ORDER BY ASC(?method)
```

Listing 4: Changing methods (CM)/classes (CC) query pattern.

class	method	cm	cc
CompareUIPlugin	getDefault()	30	10
Utilities	getString(java.util.ResourceBundle,java.lang.String)	26	14
Utilities	getString(java.lang.String)	24	12
ICompareInput	getLeft()	16	9
ICompareInput	getRight()	15	8

Table 3: Changing methods/classes for the `compare` plug-in

5.3.2. Number of methods (NOM) and number of attributes (NOA)

The queries shown in Listing 5 and 6 calculate the two metrics *number of attributes (NOA)* and *number of methods (NOM)* that can be both used as indicators for possible *God classes* (see also Section 5.4). The results are shown in Table 4(a) and 4(b) respectively. Having a closer look at class `TextMergeViewer`, one can observe that the class is indeed very large with its 4344 lines of code. Also `CompareUIPlugin` is rather big with a total number of 1161 lines of code. Without examining the classes in more detail, we hypothesize that there might be some room for refactorings, which possibly result in smaller and more easy to use classes.

```
SELECT ?class (count(distinct ?attribute) AS ?noa)
WHERE
  { ?class som:hasAttribute ?attribute }
GROUP BY ?class
ORDER BY ASC(?class)
```

Listing 5: Number of attributes (NOA) query pattern.

(a) Number of attributes (NOA) metric for the `compare` plug-in.

class	noa
TextMergeViewer	91
PatchMessages	63
CompareUIPlugin	42
ContentMergeViewer	36
CompareMessages	27
EditionSelectionDialog	26
CompareEditorInput	23
CompareConfiguration	20
ICompareContextIds	19
ComparePreferencePage	18

(b) Number of methods (NOM) metric for the `compare` plug-in.

class	nom
TextMergeViewer	115
CompareUIPlugin	46
ContentMergeViewer	44
OverlayPreferenceStore	43
Patcher	39
CompareEditorInput	38
Utilities	34
MergeSourceViewer	31
EditionSelectionDialog	30
CompareConfiguration	28

Table 4: The results of NOA and NOM queries.

```
1 SELECT ?class (count(distinct ?method) AS ?nom)
2 WHERE
3   { ?class som:hasMethod ?method }
4 GROUP BY ?class
5 ORDER BY ASC(?class)
```

Listing 6: Number of methods (NOM) query pattern.

5.3.3. Number of bugs (NOB) and number of revisions (NOR)

To close this section and to support or discard our hypothesis from the previous paragraph, we measured the *number of bug reports (NOB)* issued per class as we assume a relationship between the number of class methods (attributes) and the number of filed bug reports. To that end, we executed a query (not shown) similar to the one presented in Listing 6. Indeed, there is a relationship as the results in Table 5(a) clearly show: the two classes `TextMergeViewer` and `CompareUIPlugin` are also among the top 10 of the most buggiest classes in the project.

(a) Number of bug reports for the `compare` plug-in.

file	nob
TextMergeViewer	36
CompareEditor	16
Patcher	15
PreviewPatchPage	13
ResourceCompareInput	12
DiffTreeViewer	10
Utilities	10
CompareUIPlugin	9
StructureDiffViewer	9
PatchWizard	6

(b) Number of revision for the `compare` plug-in.

file	nor
TextMergeViewer	213
CompareEditorInput	88
CompareUIPlugin	70
ContentMergeViewer	69
EditionSelectionDialog	66
Utilities	64
CompareEditor	57
Patcher	51
ComparePreferencePage	50
DiffTreeViewer	47

Table 5: The results of NOB and NOR queries.

Finally, the *number of revisions (NOR)* metric counts the number of revisions of a file recorded in CVS. The respective results are shown in Table 5(b) (again, for space consideration we omitted the listing of the query. However, it is very similar to Listing 6). Both NOB and NOR are used in Section 5.5 to determine *defect* and *evolution density* of software

systems.

5.4. Task 3: detection of code smells

In a third set of experiments, we evaluate the applicability of SPARQL to the task of detecting *code smells* [2]. In other words, the question is whether SPARQL is able to give you a *hint* that there *might* be potential problems in the source code. Can SPARQL tell you if it could be solved, for instance, by refactoring the current architecture? In order to solve this task, we selected one candidate smell, the *GodClass* anti-pattern, which we thought could be (among others) identified in the `compare` plug-in.

Furthermore, the following experiments are useful to demonstrate the benefits of *ontological reasoning* for software analysis. We, therefore, use two well-known inference engines in the Semantic Web: the Jena[20] reasoner and the complete OWL reasoner Pellet.[21] Note that, while the Jena reasoner only supports a subset of the OWL language (*i.e.*, OWL/Lite), Pellet is complete, in other words is able to deal with all elements of the OWL/DL language. These reasoners are used to derive additional RDF assertions which are entailed from base facts together with the ontology information from the source code models (Section 3) and the axioms and rules associated with the reasoners.

5.4.1. GodClass anti-pattern

A God class is defined as a class that potentially ''knows" too much (its role in the program becomes all-encompassing). In our sense, it has (too) many methods and instance variables. In the following, we present two approaches to find God classes in source code: first, by computing object-oriented source code metrics (see Section 5.3) and, second, by inferring them using the aforementioned reasoning engines. To illustrate our approach, we define a God class as any class which declares more than 20 methods and attributes in its body.

We first present the metrics-based approach. Listing 7 shows a particular SPARQL query that counts both the number of methods (NOM) and number of attributes (NOA) per class. A God class is successfully identified if both are above 20. The topmost results of this query are shown in Table 6.

```
1  SELECT ?GodClass (count(distinct ?method) AS ?nom)
2                   (count(distinct ?attribute) AS ?noa)
3  WHERE
4    { ?GodClass som:hasMethod    ?method ;
5                som:hasAttribute ?attribute
6    }
7  GROUP BY ?GodClass
8  HAVING ( ( count(distinct ?method) > 20 )
9         && ( count(distinct ?attribute) > 20 ) )
10 ORDER BY ASC(?GodClass)
```

Listing 7: SPARQL metrics approach to find God classes.

[20] http://jena.sourceforge.net/

[21] http://pellet.owldl.com/

GodClass	nom	noa
TextMergeViewer	115	91
CompareUIPlugin	46	42
ContentMergeViewer	44	36
CompareEditorInput	38	23
EditionSelectionDialog	30	26

Table 6: Results of God class query pattern.

To demonstrate how God classes can be inferred from the ontology, the *ontological concept 'GodClass'* has to be defined first. We chose OWL as concept definition language as it offers all the required language constructs.

Figure 3 shows the definition of the ontology concept 'GodClass' in description logic syntax (DL Syntax) wich can be easily transformed to OWL syntax, *e.g.*, N3. We define a new class (`GodClass`) which is equivalent to an anonymous class of the type `som:Class` having at least 21 `hasMethod` and 21 `hasAttribute` relations.

$$\text{smell:GodClass} \equiv \text{som:Class} \sqcap \geqslant 21\ \text{som:hasMethod} \sqcap \geqslant 21\ \text{som:hasAttribute}$$

Figure 3: GodClass concept definition (in DL Syntax).

Having defined the ontological concept for a GodClass and, of course, an inferred ontology created by a reasoner, it is now possible to use the query shown in Listing 8 to find all God classes in the `compare` plug-in.

```
1 SELECT ?GodClass
2 WHERE
3   { ?GodClass a smell:GodClass }
```

Listing 8: SPARQL reasoning approach to find God classes.

5.4.2. Orphan methods

To close this section, we give an example where ontological reasoning is not successful although the concept can be perfectly defined in OWL. Figure 4 shows the logical definition for *orphan methods* (*i.e.*, methods that are not invoked by any other method in the project). The ¬ expression describes a logical negation: the class extension consists of those methods that are not invoked by any other behavioral entity (*i.e.*, any other method).

Due to the *open-world semantics* of OWL that states that if a statement cannot be inferred from the RDF data set, then it still cannot be inferred to be false, most inference engines, including the ones used in this work, are not able to find concepts of type `OrphanMethod`.

Therefore, the query shown in Listing 9 does not return any results. Fortunately, there is a trick one can do in SPARQL queries to get a little bit of *closed-world reasoning* — the ability to answer *negative* queries although the RDF data set does not contain explicit information about the absence of certain facts.

$$\boxed{\text{smell:OrphanMethod} \equiv \text{som:Method} \sqcap \neg\text{som:isInvokedBy.BehaviouralEntity}}$$

Figure 4: Orphan method concept definition (in DL Syntax).

```
1 SELECT ?orphanMethod WHERE
2   { ?orphanMethod a smell:OrphanMethod }
```

Listing 9: Orphan method query pattern.

The trick is achieved by the `bound` operator in the filter clause on line 6 in Listing 10, which returns true if its variable (`?invoker`) is bound to a value. The query in Listing 10 finds all `?orphanMethod`s, gets any `isInvokedBy`, and filters those which passed through the optional branch.

```
1 SELECT ?orphanMethod WHERE
2   { ?orphanMethod rdf:type som:Method .
3     OPTIONAL
4       { ?orphanMethod som:isInvokedBy ?invoker }
5     FILTER ( ! bound(?invoker) )
6   }
```

Listing 10: Orphan method query pattern.

However, some ontological reasoning is still required in this query, as the property `isInvokedBy` is defined as `owl:inverseOf invokes` in our software ontology model. In other words, results of the form *method1 isInvokedBy method2* must be inferred from the inverse `invokes`-statements.

The query returns numerous results of which we only present one. It finds, for instance, the public method `discardBuffer()` declared on class `BufferedContent`. This method is never invoked by any other class in the `compare` plug-in. Orphan methods could possibly be removed from the interface of a class without affecting the overall functionality of the system to result in a more clean and easy to understand source code.

5.5. Task 4: defect and evolution density

With our next set of experiments, we aim at determining a file's as well as a whole software project's *Defect* and *Evolution Density*. Note that in this context, we consider files as "containers" for classes and instance variables (*i.e.*, they may contain multiple classes as well as inner classes). Inspired by Fenton [41], defect density DED_f of a file f is defined as the ratio of the number of bug reports (NOB) over the total number of revisions (NOR) of f, *i.e.*,

$$DED_f = \frac{NOB}{NOR} \tag{1}$$

where NOB and NOR are the metrics presented in Section 5.3. Next, we define a file's/project's *Evolution Density* as counterpart to defect density. When we refer to evolution density, we think of all the changes made to a software system which were not bug-fixing,

but "ordinary" software development, such as functional extension and improvement, adaption, and testing. The evolution density EVD_f of a file f is, therefore, defined as:

$$EVD_f = 1 - DED_f \tag{2}$$

Table 7 lists evolution and defect density for the 5 topmost classes of the `org.eclipse.compare` plug-in in descending order of defect density retrieved with the query shown in Listing 11. Visualizing the defect density (Figure 5(a)) brings to light some interesting facts: first, only about 25% of all source files contain bugs at all. Nearly 75% of the code is free of defects (measured by the reported bugs); second, the concentration of the errors is exponentially decreasing (*i.e.*, only few files have a high concentration of bugs). This is further illustrated in Figure 5(b), which shows a histogram of the number of classes in the project per 0.1 DED interval.

```
1  SELECT ?fileName (count(?revision) AS ?NOR)
2   (count(?bug) AS ?NOB)
3   (count(?bug)/count(?revision) AS ?DED)
4   (1-count(?bug)/count(?revision) AS ?EVD)
5  WHERE {
6   ?file vom:hasRevision ?revision .
7   ?file vom:name ?fileName .
8   OPTIONAL{
9    ?bug bom:hasResolution ?revision .
10  }
11  FILTER(regex(?fileName, "\\.java$" , "i")) .
12  }
13  GROUP BY (?fileName)
```

Listing 11: Evolution and density query pattern.

File	NOR	NOB	EVD	DED
StatusLineContributionItem.java	3	3	0.000	1.000
CompareNavigator.java	3	2	0.333	0.667
IResourceProvider.java	4	2	0.500	0.500
DifferencesIterator.java	10	5	0.500	0.500
PatchProjectDiffNode.java	2	1	0.500	0.500

Table 7: Evolution and defect density of the `org.eclipse.compare` plug-in.

Finally, to calculate measures *over all software engineering activities* in the project, *Total Evolution Density* ($TEVD$) and *Total Defect Density* ($TDED$) are defined as shown in Equations 3 and 4 (with n being the number of files).

$$TEVD = \frac{\sum_{f=1}^{n} EVD_f}{n} \tag{3}$$

$$TDED = \frac{\sum_{f=1}^{n} DED_f}{n} = 1 - TEVD \tag{4}$$

For the `org.eclipse.compare` plug-in release 3.2.1, the value for $TDED$ is 0.054, which expresses that 5.4% of all activities in the project are due to bug-fixing and 94.6% due to

(a) DED per file. (b) DED histogram.

Figure 5: The figures show the defect density *DED* per file and the number of classes per 0.1 DED interval in the `org.eclipse.compare` plug-in release 3.2.1.

functional extension (among others). These findings seem to disagree with those of Boehm [42] who found that approximately 12% of all software engineering tasks are bug-fixing. We hypothesize that the *time span* of the measurements and the *bug reporting discipline* are reasons for this divergence in results and postpone it to future work to prove or reject this hypothesis.

5.6. Task 5: bug prediction

For our final bug prediction experiments, we will use our SPARQL-ML approach (SPARQL Machine Learning) – an extension of SPARQL that extends the Semantic Web query language with knowledge discovery capabilities (see Section 4.2). In order to show the usefulness of SPARQL-ML for bug prediction, we repeated the *defect location experiment* presented in [43]. The goal of this experiment was to predict the probability of defect (bug) occurrence for any given file from a test set given an induced model from a training set. The data for the experiment was collected from six plug-ins of the Eclipse open source project as in [43]: `updateui`, `updatecore`, `search`, `pdeui`, `pdebuild`, and `compare`.

The experimental procedure can be summarized as follows: first, along with the data from CVS and Bugzilla, we exported each plug-in into our Semantic Web EvoOnt format

[6][22]; second, providing a small extension to EvoOnt, we took into account the extra features from [43] that are used for learning and predicting; and third, we wrote SPARQL-ML queries for the induction of a mining model on the training set as well as for the prediction of bugs on the test set. The queries for both tasks are shown in Listings 12 and 13 respectively.

In the past, many approaches have been proposed to perform bug prediction in source code. In Fenton and Neil [44], an extensive survey and critical review of the most promising learning algorithms for bug prediction from the literature is presented. They proposed to use *Bayesian Belief Networks (BBNs)* to overcome some of the many limitations of the reviewed bug prediction algorithms. It is imporatant to note that the relational Bayesian classifier (RBC) validated in this case study is an extension of the naïve Bayesian classifier (that applies Bayes' rule for classification) to a relational data setting.

```
1  CREATE MINING MODEL <http://www.example.org/bugssignificant>
2    { ?file                    RESOURCE    TARGET
3      ?error                   DISCRETE    PREDICT  {'YES','NO'}
4      ?lineAddedIRLAdd         CONTINUOUS
5      ?lineDeletedIRLDel       CONTINUOUS
6      ?revision1Month          CONTINUOUS
7      ?defectAppearance1Month  CONTINUOUS
8      ?revision2Months         CONTINUOUS
9      ?reportedIssues3Months   CONTINUOUS
10     ?reportedIssues5Months   CONTINUOUS
11   }
12 WHERE
13   { ?file      vom:hasRevision   ?revision .
14     ?revision  vom:creationTime  ?creation .
15     FILTER (xsd:dateTime(?creation) < "2007-01-31T00:00:00"^^xsd:dateTime)
16
17     ?file      vom:hasError  ?error .
18
19     OPTIONAL { ?file  vom:lineAddedIRLAdd         ?lineAddedIRLAdd . }
20     OPTIONAL { ?file  vom:lineDeletedIRLDel       ?lineDeletedIRLDel . }
21     OPTIONAL { ?file  vom:revision1Month          ?revision1Month . }
22     OPTIONAL { ?file  vom:defectAppearance1Month  ?defectAppearance1Month . }
23     OPTIONAL { ?file  vom:revision2Months         ?revision2Months . }
24     OPTIONAL { ?file  vom:reportedIssues3Months   ?reportedIssues3Months . }
25     OPTIONAL { ?file  vom:reportedIssues5Months   ?reportedIssues5Months . }
26   }
27 USING <http://kdl.cs.umass.edu/proximity/rpt>
```

Listing 12: SPARQL-ML model induce statement.

The results are illustrated in Figure 6, showing the results in terms of prediction accuracy (acc; in legend), Receiver Operating Characteristics (ROC; graphed), and the area under the ROC-curve (auc; also in legend). The ROC-curve graphs the true positive rate (y-axis) against the false positive rate (x-axis), where an ideal curve would go from the origin (0,0) to the top left (0,1) corner, before proceeding to the top right (1,1) one [45]. It has the advantage to show the prediction quality of a classifier independent of the distribution of the underlying data set (e.g. the skewed ratio between bug and no-bug). The area under the

[22]Information from CVS and Bugzilla was considered from the first releases up to the last one released in January 2007.

```
SELECT DISTINCT ?file ?prediction ?probability
WHERE
  { ?file      vom:hasRevision   ?revision .
    ?revision  vom:creationTime  ?creation .

    FILTER (xsd:dateTime(?creation) <= "2007-01-31T00:00:00"^^xsd:dateTime)

    OPTIONAL { ?file  vom:lineAddedIRLAdd         ?lineAddedIRLAdd . }
    OPTIONAL { ?file  vom:lineDeletedIRLDel       ?lineDeletedIRLDel . }
    OPTIONAL { ?file  vom:revision1Month          ?revision1Month . }
    OPTIONAL { ?file  vom:defectAppearance1Month  ?defectAppearance1Month . }
    OPTIONAL { ?file  vom:revision2Months         ?revision2Months . }
    OPTIONAL { ?file  vom:reportedIssues3Months   ?reportedIssues3Months . }
    OPTIONAL { ?file  vom:reportedIssues5Months   ?reportedIssues5Months . }

    PREDICT
      { ( ?prediction ?probability )
              sml:predict  (
                      <http://www.example.org/bugssignificant>
                ?file ?lineAddedIRLAdd ?lineDeletedIRLDel
                ?revision1Month ?defectAppearance1Month ?revision2Months
                ?reportedIssues3Months ?reportedIssues5Months ) .
      }
  }
```

Listing 13: SPARQL-ML predict statement.

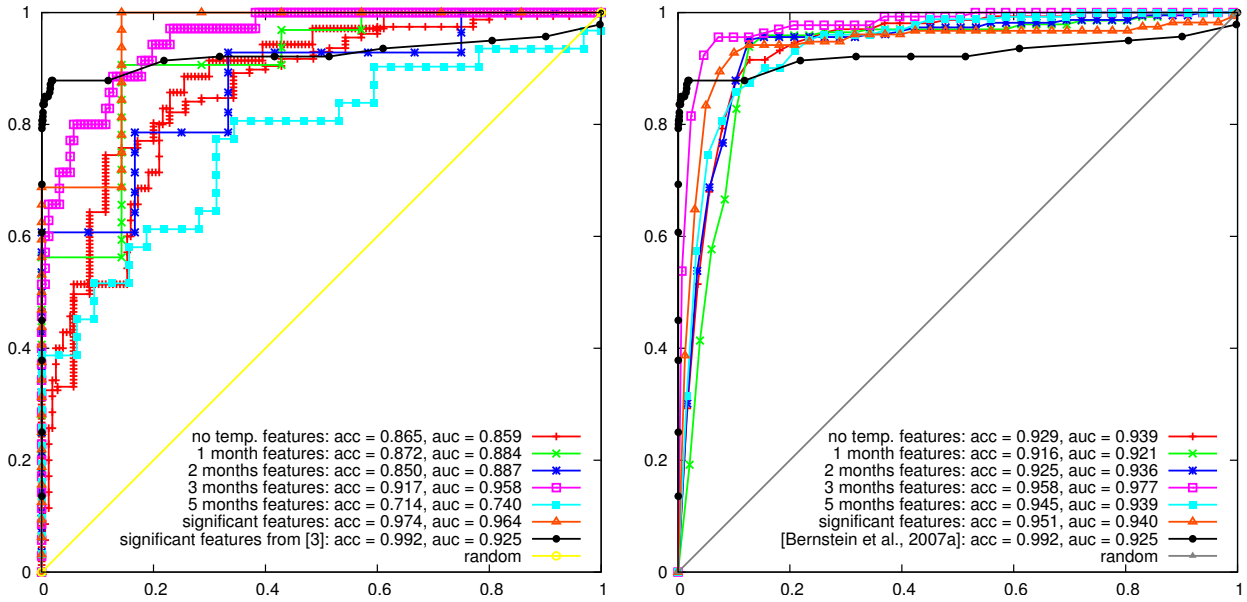
ROC-curve is, typically, used as a summary number for the curve. An in-depth explanation about ROC-curves can be found in [46]. Note that this experiment clearly illustrates the simplicity by which the experiment from [43] can be reduced to running an off-the-shelf query.

6. Conclusions, limitations, and future work

In this paper, we presented a novel approach to analyze software systems using Semantic Web technologies. As exemplified by the case studies above EvoOnt provides the basis for representing software source code and meta-data in OWL. This representation allows to reduce many mining software repository tasks to simple queries in the Semantic Web query language SPARQL (and its extensions iSPARQL and SPARQL-ML).

This format is principally used within the Semantic Web to share, integrate, and reason about data of various origin. We evaluated the use of the format in the context of analyzing the `org.eclipse.compare` plug-in for Eclipse.

To illustrate the power of using EvoOnt we conducted five sets of experiments in which we showed, firstly, that it was expressive enough to shed some light on the evolution of software systems (especially when using iSPARQL and its imprecise querying facilities); secondly, that it allowed to find code smells, hence, fosters refactoring; thirdly, that it enables the easy application of software design metrics to quantify the size and complexity of software; forth, that it, due to OWL's ontological reasoning support, furthermore allows to derive additional assertions, which are useful for software engineering tasks; and fifth, that it enables defect and evolution density measurements expressing the amount of bug-fixing and "ordinary" software development as measured by all software engineering tasks.



(a) RPT performance. (b) RBC performance.

Figure 6: ROC-curves to show a performance comparison of the two classifiers *Relational Probability Tree (RPT)* and *Relational Bayesian Classifier (RBC)*.

A limitation of our approach is the loss of information due to the use of our FAMIX-based software ontology model. Language constructs such as switch-statements are not modeled in our ontology. Hence, the effects are that measurements on the statement level of source code cannot be conducted. This limitation can be addressed by adding elements from additional meta-models such as DMM [27].

Also, one of the greatest impediments towards the widespread use of EvoOnt is the current lack of high-performance industrial-strength triple-stores and reasoning engines. Without such engines most software developers are likely to retort to relational storage solutions that are ill-suited for storing these graph-like data [40, 39]. Some newer developments both in industry[23] and in academe [40, 39] are encouraging. They indicate that fast engines are possible and likely to become more available in the near future. With their advent the widespread use of the techniques proposed here will become feasible and attractive to software developers.

In summary, we have shown that the use of EvoOnt can simplify a large number of software engineering tasks attempted by the mining software repositories community. We think that approaches like EvoOnt have an even greater potential as they would foster more exchange leading to a better integration of results between different analyses or simplifying inter-software-project software analyses — a problem so far avoided by most researchers due to the complexities of integrating the data. Also, the choice of the OWL as the underlying knowledge representation simplifies the extension of the model with other sources such as

[23] *e.g.* AllegroGraph, `http://agraph.franz.com/allegrograph/`

data extracted mailing lists or run-time properties of the code.

Acknowledgements

We would like to thank the participants of the 2007 ICSE Mining Software Repositories workshop and the 2007 ESWC Semantic Web Software Engineering Workshops for their valuable comments on our earlier work. We would also like to thank the anonymous reviewers for their invaluable comments that helped improve the paper.

[1] T. C. Lethbridge, J. Singer, A. Forward, How Software Engineers Use Documentation: The State of the Practice, IEEE Software 20 (6) (2003) 35–39.
[2] M. Fowler, Refactoring, Addison-Wesley, 1999.
[3] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice, Springer, 2006.
[4] C. Kiefer, A. Bernstein, M. Stocker, The Fundamentals of iSPARQL — A Virtual Triple Approach For Similarity-Based Semantic Web Tasks, in: Proceedings of the 6th International Semantic Web Conference (ISWC), 2007.
[5] C. Kiefer, A. Bernstein, A. Locher, Adding Data Mining Support to SPARQL via Statistical Relational Learning Methods, in: Proceedings of the 5th European Semantic Web Conference (ESWC), Springer, 2008.
[6] C. Kiefer, A. Bernstein, J. Tappolet, Mining Software Repositories with iSPARQL and a Software Evolution Ontology, in: Proceedings of the 4th ICSE International Workshop on Mining Software Repositories (MSR), 2007.
[7] C. Kiefer, A. Bernstein, J. Tappolet, Analyzing Software with iSPARQL, in: Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE), Springer, 2007.
[8] H. A. Müller, K. Klashinsky, RIGI – A System for Programming-in-the-large, in: Proceedings of the 10th international Conference on Software Engineering (ICSE), 1988.
[9] Y.-F. Chen, M. Nishimoto, C. Ramamoorthy, The C Information Abstraction System, IEEE Transactions on Software Engineering 16 (3) (1990) 325–334.
[10] P. Devanbu, GENOA - A Customizable, Front-end Retargetable Source Code Analysis Framework, ACM Transactions on Software Engineering and Methodology 8 (1999) 177–212.
[11] J. Ebert, B. Kullbach, V. Riediger, A. Winter, GUPRO: Generic Understanding of Programs – an Overview, Electronic Notes in Theoretical Computer Science.
[12] P. Devanbu, R. Brachman, P. G. Selfridge, LaSSIE: a Knowledge-based Software Information System, Communications of the ACM 34 (1991) 34–49.
[13] OMG, Query / View / Transformation (QVT), Tech. rep., Object Management Group (2008).
URL `http://www.omg.org/spec/QVT/1.0/`
[14] R. C. Holt, A. Winter, A. Schürr, GXL: Toward a Standard Exchange Format, in: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE), 2000.
[15] R. C. Holt, Structural manipulations of software architecture using tarski relational algebra, in: Proceedings of the Working Conference on Reverse Engineering (WCRE), IEEE Computer Society, 1998, p. 210.
[16] L. Feijs, R. Krikhaar, R. Van Ommering, A relational approach to support software architecture analysis, Software-Practice and Experience 28 (4) (1998) 371–400.
[17] A. Schürr, A. J. Winter, A. Zündorf, The PROGRES Approach: Language and Environment, Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools (1999) 487–550.
[18] OMG, XML Metadata Interchange (XMI), Tech. rep., Object Management Group (1998).
URL `ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf`
[19] J. Dietrich, C. Elgar, A formal description of design patterns using OWL, in: Proceedings of the Australian Software Engineering Conference, Brisbane, Australia, 2005.

[20] E. Gamma, R. Helm, R. E. Johnson, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, 1995.
[21] D. Hyland-Wood, D. Carrington, S. Kapplan, Toward a Software Maintenance Methodology using Semantic Web Techniques, in: Proceedings of the 2nd ICSM International Workshop on Software Evolvability (SE), 2006, pp. 23–30.
[22] S. Demeyer, S. Tichelaar, P. Steyaert, FAMIX 2.0—the FAMOOS inf. exchange model, Tech. rep., University of Berne, Switzerland (1999).
URL http://www.iam.unibe.ch/~famoos/FAMIX/Famix20/Html/famix20.html
[23] R. Witte, Y. Zhang, J. Rilling, Empowering Software Maintainers with Semantic Web Technologies, in: In Proceedings of the 4th European Semantic Web Conference (ESWC), 2007.
[24] M. Mäntylä, J. Vanhanen, C. Lassenius, A Taxonomy and an Initial Empirical Study of Bad Smells in Code, in: Proceedings of the International Conference on Software Maintenance (ICSM), 2003, pp. 381–384.
[25] R. Shatnawi, W. Li, A Investigation of Bad Smells in Object-Oriented Design Code, in: Proceedings of the 3rd International Conference on Information Technology: New Generations, 2006, pp. 161–165.
[26] H.-J. Happel, A. Korthaus, S. Seedorf, P. Tomczyk, KOntoR: An Ontology-enabled Approach to Software Reuse, in: Proceedings of the 18th Internatinal Conference on Software Engineering and Knowledge Engineering (SEKE), San Francisco, CA, 2006.
[27] T. C. Lethbridge, S. Tichelaar, E. Ploedereder, The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering, in: Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering, 2004.
[28] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, Tech. rep., W3C (2008).
URL http://www.w3.org/TR/rdf-sparql-query/
[29] V. I. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions and Reversals, Soviet Physics Doklady 10 (1966) 707–710.
[30] T. Sager, A. Bernstein, M. Pinzger, C. Kiefer, Detecting similar java classes using tree algorithms, in: Proceedings of the 3rd ICSE International Workshop on Mining Software Repositories (MSR), 2006, pp. 65–71.
[31] G. Valiente, Algorithms on Trees and Graphs, Springer, 2002.
[32] J. Neville, D. Jensen, L. Friedland, M. Hay, Learning Relational Probability Trees, in: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2003, pp. 625–630.
[33] J. Neville, D. Jensen, B. Gallagher, Simple Estimators for Relational Bayesian Classifiers, in: Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM), 2003, pp. 609–612.
[34] S. Džeroski, Multi-relational data mining: An introduction, ACM SIGKDD Explorations Newsletter 5 (1) (2003) 1–16.
[35] M. Fischer, M. Pinzger, H. Gall, Populating a Release History Database from Version Control and Bug Tracking Systems, in: Proceedings of the International Conference on Software Maintenance (ICSM), Amsterdam, Netherlands, 2003, pp. 23–32.
[36] G. Gröner, S. Staab, A. Winter, Graph Technology and Semantic Web in Reverse Engineering – A Comparison, in: Proceedings of ICPC 2008 Workshop: Semantic Technologies in System Maintenance, 2008.
[37] A. Bernstein, C. Kiefer, M. Stocker, OptARQ: A SPARQL Optimization Approach based on Triple Pattern Selectivity Estimation, Tech. rep., Department of Informatics, University of Zurich (2007).
URL www.ifi.uzh.ch/pax/web/uploads/pdf/publication/143/ifi-2007_03.pdf
[38] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, D. Reynolds, SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation, in: Proceedings of the 17th International World Wide Web Conference (WWW), 2008.
[39] C. Weiss, P. Karras, A. Bernstein, Hexastore: Sextuple Indexing for Semantic Web Data Management, in: Proc. of the 34th Intl Conf. on Very Large Data Bases (VLDB), 2008.
[40] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Scalable Semantic Web Data Management Using

Vertical Partitioning, in: 33rd International Conference on Very Large Data Bases (VLDB), 2007.
[41] N. E. Fenton, Software Metrics: A Rigorous Approach, Chapman & Hall, Ltd., 1991.
[42] B. W. Boehm, Software Engineering Economics, Prentice Hall, 1981.
[43] A. Bernstein, J. Ekanayake, M. Pinzger, Improving Defect Prediction Using Temporal Features and Non-linear Models, in: Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), 2007.
[44] N. Fenton, M. Neil, A Critique of Software Defect Prediction Models, IEEE Transactions On Software Engineering 25 (3).
[45] F. J. Provost, T. Fawcett, Robust Classification for Imprecise Environments, Machine Learning 42 (3) (2001) 203–231.
[46] I. H. Witten, E. Frank, Data Mining : Practical Machine Learning Tools and Techniques, 2nd Edition, Elsevier, Morgan Kaufman, 2005.