Semantics-aware Software Project Repositories

Jonas Tappolet

Department of Informatics, University of Zurich, Switzerland
tappolet@ifi.uzh.ch

Abstract. This proposal explores a general framework to solve software analysis tasks using ontologies. Our aim is to build semantically annotated, flexible, and extensible software repositories to overcome data representation, intra- and inter-project integration difficulties as well as to make the tedious and error-prone extraction and preparation of meta-data obsolete. We also outline a number of practical evaluation approaches for our propositions.

1 Research Problem

In *Software Engineering*, many tools have been used for years to support the collaboration of development teams. Among others, these tools are typically a version control system holding the project's files in a versioned manner and a bug tracking system in which defects and enhancement requests are stored. Research has shown that these repositories contain a huge amount of additional information that can be exploited to enhance the quality of software systems (such as the detection of error-prone software patterns or the prediction of the number of defects).An in-depth analysis reveals, however, that there are three main obstacles to seamless software analysis: (1) *data representation* — the natural structure of the contents of these repositories is a typed graph (*e.g.*, source code syntax trees). However, most of current software analysis tools use relational database management systems requiring a transformation of the data to the relational format. More importantly, they rely on propositional (i.e., one table) representations for analysis, as most data mining algorithms use this format. Moreover, the data is often extracted in a form that is suitable for only a particular task. These transformations are tedious, error-prone, and, usually, lossful. (2) *Intra-project repository integration* — each of these repositories is designed as a stand-alone system covering only a specific part of the software development process. In order to generate uniform views on software projects, methods are needed to overcome the boundaries of each isolated repository. (3) *Inter-project repository integration* — fewest software projects use solely their own code. Developers make pervasive use of components and frameworks hosted in remote repositories, weaving a world-wide call graph. Hence, repositories of different projects need to be accessed in a integrated manner.

In this paper we present our EvoOnt approach to address these three problem areas. By integrating different repositories (data sources) using Semantic Web technologies, we end-up in a graph-based approach that is capable of handling distributed and heterogeneous software project data.

2 Related Work

In this section, we summarize a small selection of the most closely related studies addressing the identified problem areas.

Graph based software analysis. Collberg *et al.* [4] present Gevol, a graph-based visualization tool for CVS and Java. The aim is to support developers understanding the software by providing visual representations of the source code and its history. Sager *et al.* [13] present *Coogle* (Code Google) that implements a set of tree similarity measures to detect similarities between Java classes of different releases of software projects. Coogle's approach is to first transform the abstract syntax tree (AST) representations of Java classes into intermediary FAMIX tree representations [6], and second, to measure their similarity by applying tree similarity algorithms. FAMIX is a software source code meta model designed to serve as an exchange format for object-oriented programming languages using flat text streams. Dietrich [7] proposed an OWL ontology to model the domain of software design patterns to automatically generate documentation about the patterns used in a software system. With the help of this ontology, the presented pattern scanner inspects the ASTs of source code fragments to identify the patterns used in the code.

baetle is an open-source project that aims to add semantics to software repositories with a strong emphasis on bug tracker data.[1] We tightly work together with the *baetle* developers to combine our ontologies with theirs.

Intra-project repository integration. Fischer *et al.* [8] present their *Release History Database* (RHDB) that integrates versioning and bug tracking systems. The data is extracted and stored in a traditional relational database.

Antoniol *et al.* [1] combined the relational RHDB with FAMIX to integrate source code with bug tracking and versioning system information.

Inter-project repository integration. Chang and Mockus [3] present a method to detect file copies among different versioning systems to build a unified version history.

None of the approaches above combines the strengths of an integrated software ontology to address all three obstacles identified in the introduction.

3 Contributions

Data representation. Most of the studies presented in Section 2 use flat representations of source code. This forces analyses to be on a textual level. Although, valuable information can be extracted using text mining techniques, it is generally hard to detect different types of source code changes (*e.g.*, structural vs. nominal changes). Consider a similarity measure which is able to find changes on the textual source code level (i.e., treating software code as a string of characters). Although the measure can, for instance, say that two software artifacts are different if their copyright notes have changed, it can, however, not say anything about the impact of this change on the software's functionality. Therefore, we use

[1] `http://code.google.com/p/baetle/`

a graph-based approach to model the repositories using RDF/OWL ontologies, which allows both textual and structural analyses.

Intra-project repository integration. There are two different relation types among repositories. *Implicit* connections are defined by the data itself or given by the nature of a repository. The relation between a file's meta-data (e.g., create date, file name, version information) and its content implicitly connects different versions of the source code. *Explicit* connections need to be manually defined. The connection between a bug report (*i.e.*, a bug fix typically reported in a bug reporting tool such as Bugzilla) and a specific file version (typically stored in a version control systems such as CVS) needs to be explicitly defined by a developer. This is usually done by mentioning a bug number in the comment of a file's new version (commit message), which can be extracted using simple text mining techniques [8] to link the respective bug report with a version of a file. However, the extractability of such explicit connections relies on disciplined and uniform reporting practices of a development team. Another method of linking a bug with a version is to compare the closing date of a bug report with the creation date of a new version. Having matching or near-matching dates is a strong indicator for a connection.

With the integration of repositories we can access the history of a file with all changes made during a file's life cycle. We can differentiate between evolutionary changes (extension of functionality) and maintenance changes (fixes of bugs).

Inter-project repository integration. In a next step we can integrate a software project's model with the models of used external components. Whenever a program makes a call to an entity that is not located inside the same project, this can be considered an external function call. Our aim is to map these calls to their representing source code model in a remote repository. One convenient method is to relate external calls in the same way as internal ones differentiating them only using their differing namespaces (which need to have a uniform transformation to the source-code-namespace/package-declaration). Having this integration, we can explore the dependencies between a software and its components analyzing, for example, the impact of a component's replacement with another (e.g., How does the code need to get adapted?) or the relation between bugs and the usage of external components.

4 Research Plan

4.1 Current State of our Research

In a first step, we implemented a set of tools to extract and interconnect data from software repositories (*i.e.*, CVS, Bugzilla, and Java), and store it as instances according to EvoOnt's model. So far, we conducted several experiments using query techniques, reasoning, similarity measures, and machine learning to evaluate EvoOnt's ability to serve as a general software analysis framework. We briefly summarize our conducted experiments. In our previous work [11, 10], the experiments are described in detail with example data from the Eclipse project.

Software metrics. We used plain SPARQL queries to compute object-oriented software metrics [12]. These metrics are, *e.g.*, the number of bugs per file, the relative number of bugs or the fan-in fan-out of a class.

Software pattern detection. Using ontological reasoning, we are able to detect software patterns as well as anti-patterns, and code smells [12]. We achieved this by defining a pattern (anti-pattern) using the concepts from our EvoOnt. We build up an own pattern ontology which can, when conducting pattern detection, be combined with the existing ontologies an a reasoner will then match the defined patterns in the data.
Similarity measures / Software evolution. Having the data in a graph-based format, we are able to calculate structural similarities between two versions of a source code file using iSPARQL [9] running similar analyses as Coogle by executing a single iSPARQL query.
Machine learning. Using SPARQL-ML, a SPARQL extension with machine learning algorithms, we were able to simply reproduce tedious bug prediction analyses [2].

4.2 Next Steps

Intra-project integration. So far, we used Bugzilla-, CVS-, and Java-repositories as data sources to extract software information from. However, there are various other products, we plan to integrate: Jira (bug tracker), Subversion (verisoning system), and C# (programming language) are our next candidates to write RDF/OWL interfaces for. On the other hand, there are other repository types involved in the software development process such as mailing-lists and forums which we plan to integrate as well into our unified framework. These types reflect the social network structure around the development process.
Inter-project integration. Inspired from Data-Warehousing, where heterogeneous data is accessed through a sole interface, we plan to implement such behavior in EvoOnt as well. With the implementation of web-based, RDF/OWL enabled interfaces exposing SPARQL-endpoints (*e.g.*, for versioning systems), it would be able to execute and answer SPARQL queries over the web. This enables a repository to be linked from any other software project using this component's functionality.
Integration with software project repositories. We intend to integrate the semantic capabilities of EvoOnt into commonly used software project repository tools (such as Subversion and Jira) making the tedious extraction and preparation of meta-data obsolete.
Evaluation. In our first set of experiments, we evaluated EvoOnt against a wide variety of software analysis tasks. In a next step, we plan to deepen certain experiments. A first selection is:
The identification of the location of a bug. Usually, a developer links a version of a source code file with a bug report whenever she fixed a specific bug. Derived from this information, we can use graph algorithms to compute deltas between the fixed and the pre-fixed source code version resulting in a subgraph exactly representing the change made for fixing a bug. Having this portion of change, we can try to identify the point in the history of the software when this changed code fragment was inserted or modified. Our hypothesis would be that this is the point in the software history where the bug was introduced.
Analyze the code co-evolution of projects and their components. This important

task has been very difficult so far, as the inter-project connections has been largely missing. The inter-project integration allows to uncover the relation between bugs of different projects. A bug may be misleadingly reported in a project due to a bug in the referenced project. Moreover, the impact of updating to a new version of a component that includes several bug fixes, and may have changed its behavior can be made visible.
Analyze the connection between code elements and people with respect to their relationship to Conway's Law [5] and perform other in-depth social network analyses.

5 Conclusions

This proposal outlined the advantages of applying Semantic Web technologies to the field of Software Analysis. Specifically, we discussed how Semantic Web technologies allow to overcome the data representation, intra-, and inter-project repository integration problems. We, furthermore, succinctly outlined how we intend to evaluate our approach by showing its usefulness for a variety of software analysis tasks and publish the findings in the software engineering literature. We also indicated our plans to develop semantically annotated software repositories, which will make the extraction and preparation of meta-data obsolete.

References

1. G. Antoniol, M. D. Penta, H. C. Gall, and M. Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *ENTCS*, 2005.
2. A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. *IWPSE*, 2007.
3. H.-F. Chang and A. Mockus. Constructing universal version history. *MSR*, 2006.
4. C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. *SoftVis*, 2003.
5. M. Conway. How do committees invent? *Datamation*, 1968.
6. S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. 1999.
7. J. Dietrich and C. Elgar. A formal description of design patterns using owl. *ASWEC*, 2005.
8. M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. *ICSM*, 2003.
9. C. Kiefer, A. Bernstein, and M. Stocker. The fundamentals of iSPARQL - a virtual triple approach for similarity-based semantic web tasks. *ISWC*, 2007.
10. C. Kiefer, A. Bernstein, and J. Tappolet. Analyzing software with iSPARQL. *SWESE*, 2007.
11. C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with iSPARQL and a software evolution ontology. *MSR*, 2007.
12. M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.
13. T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar java classes using tree algorithms. *MSR*, 2006.