Clone Detection Meets Semantic Web-Based Transitive Closure Computation

Iman Keivanloo and Juergen Rilling
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
{i_keiv, rilling@cse.concordia.ca}

***Abstract*—In this paper we discuss a new application of Semantic Web and Artificial Intelligence in software analysis research. We show on a concrete example - clone detection for object-oriented source code that transitivity closure computation can provide added value to the clone detection community. Our novel approach models the domain of discourse knowledge as a mixture of source code patterns and inheritance trees represented as Directed Acyclic Graphs. Our approach promotes the use of Semantic Web and inference engines in source code analysis. More specifically we take advantage of the Semantic Web and its support for knowledge modeling and transitive closure computation to detect semantic source code clones not detected by traditional detection tools.**

***Keywords*-Clone detection; object oriented; Semantic Web**

I. INTRODUCTION

The goal of source code clone detection is to find similar code fragments (clone-pairs) by applying some form of pattern matching on software artifacts. The degree of pattern matching is typically expressed by the clone type [1, 2] being detected. Type-1 clones are code fragments (e.g., lines) that are exactly identical except for white spaces. Type 2 clones additionally can contain minor changes due to renaming (e.g. a variable name has been changed). Type-3 clones cover also major modifications to code fragments, such as additional statements. Nevertheless, these clone-pairs still have a syntactically or semantically resemblance.

Existing clone detection approaches range from simple generic textual pattern matching approaches (e.g. Diff [3]) without considering source code semantics. At the other end of the spectrum, there are specialized clone detection approaches, developed by the clone detection research community to take advantage of source code semantics. Abstract Syntax Tree (AST) [4], token sequences [5], Program Dependency Graph (PDG) [6], and quality metrics (e.g. LOC) [7], form the basis for most of these specialized clone detection approaches.

In [1, 2, 8] it has been discussed that the context in which a clone detection approach is being used (e.g. plagiarism detection, refactoring) plays a significant role in evaluating its quality. Depending on the application context, not only the available information sources (artifacts being analyzed) but also what constitutes a clone will differ.

Object Oriented programming principles, such as the *Liskov substitution principle*, provide many potential application contexts for clone detection during software maintenance (e.g. refactoring, reuse). For example, consider the three code fragments in Fig. 1. All three fragments share similar code patterns except their class names. If we apply clone detection in a plagiarism or bad code smell detection context, these three code fragments should be detected and classified as a single clone group. However, when one considers a refactoring context, additional information about the source code semantics should be used during the detection. For example, using the "Pull-Up Method" refactoring pattern [8, 9], where a parent class has several child classes which contain the same method (code clones), pulling up the method to the parent can remove these clones. For such clone application contexts, the detection precision (reduction of false positives) is of essence.

One approach to improve precision is by taking advantage of program semantics as an additional source of information to detect and distinguish not only syntactical but also *semantic similar clones.* Given the availability of a second metadata information source (e.g. an inheritance tree in Fig. 1 right column) during pattern matching, one can now detect for example in Fig.1: a.) fragment #1 and #2 are based on the same class hierarchy and therefore should be considered as a clone group and b.) fragment #3, while containing a similar code fragment, is not actually part of this inheritance hierarchy and should therefore not be considered part of clone group for code fragment #1 and #2.

Opportunity. To the best of our knowledge, there has been no algorithm for automated clone detection where *inheritance tree* semantics is used as part of the clone detection process. The lack of such approach constitutes our major motivation to devise a clone detection and search approach which not only takes into consideration the pattern similarity but also the inheritance tree semantics of the object oriented source code.

Solution. The Semantic Web has been devised as an open and general purpose reasoning enabled infrastructure. It is based on best practices from Artificial Intelligence and Web technologies. Thus, it is able to reason about any type of formally represented knowledge via inference engines. The only restriction is to model the domain of discourse using its knowledge representation languages such as OWL. In this position paper, we discuss potential synergies between Semantic Web technologies and traditional source code analysis techniques. As a concrete (running) example we illustrate how the Semantic Web can provide added value to the clone detection community. In this example, we take advantage of transitive closure querying and reasoning to support the detection of semantic clones.

978-1-4673-1753-5/12/$31.00 ©

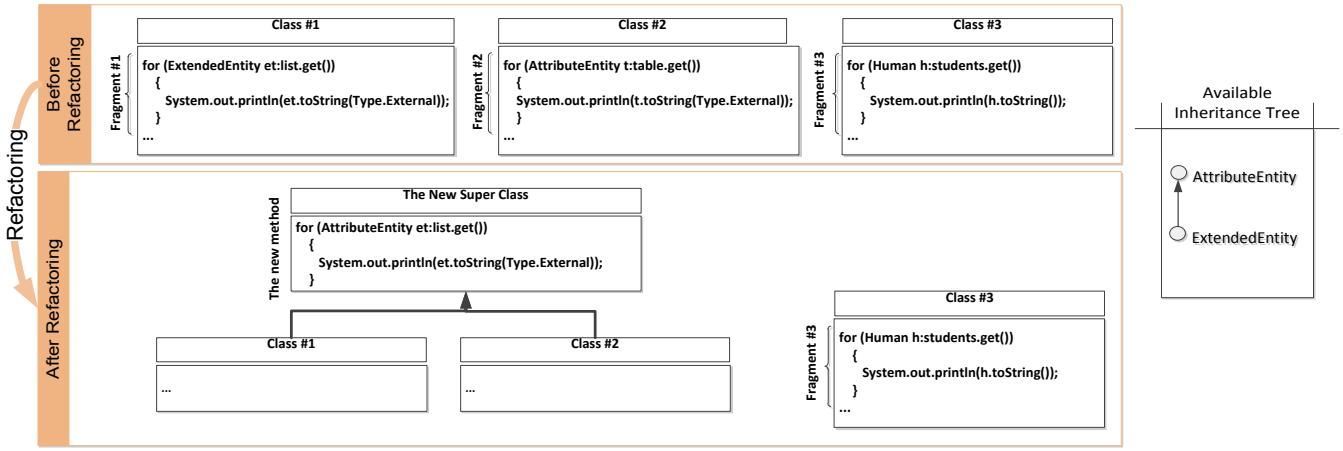

Figure 1. A potential case for Semantic Web-based reasoning to improve clone detection usability.

The remainder of this paper is organized as follows: Section 2 introduces inheritance trees as an important source of information for clone detection. Section 3 covers transitive closure and its use in clone detection. Section 4 illustrates the use of Semantic Web in clone detection, followed by conclusions and future work in Section 5.

II. Inheritance Trees in Clone Detection

In this section, we discuss how meta information in the form of an inheritance tree can benefit clone detection. In OO, accessed types (i.e. classes or interfaces) are essential to determine the functionality and similarity relationships of code fragments. At source code level, the type fingerprints are less explicit, since determining a variable names' fingerprint requires the availability of semantic information. For example, in Fig. 1, the type of variable `students` is not explicitly specified. This is contrast to the binary level, where such type fingerprints are made explicit. Fig. 2 shows two method blocks taken from the jgoodies project (jgoodies.com). In this example, the only dissimilarity at source code level is the *enumeration token* (i.e. Orange and Yellow constants) which is negligible (Fig. 2-A). Based on our earlier discussion, we are interested in providing a clone detection approach that is sensitive to type usage for maintenance tasks (e.g. refactoring). When comparing the binary file content additional dissimilar sections (not present in the corresponding source code) can be detected. This dissimilarity is highlighted in Fig. 2-C (which summarizes the accessed Java types within each fragment for illustration purpose). In this example, the PlasticTheme and SkyBluerTahoma types constitute the major dissimilarity among the method blocks. Apparently, this type of fingerprint dissimilarity is not explicitly visible at the source code and only at binary. However, since the binary information is typically available, it can be exploited to improve the precision of the clone detection.

For our approach we take advantage of the relationship between the two Java classes (PlasticTheme and SkyBluerTahoma), to determine their degree of similarity. We consider classes with some form of inheritance relationship to be closer related than classes without any inheritance relationship (similar to the example in Fig. 1). In Fig. 2 (Section D), we illustrate different types of inheritance relationships among two classes. This additional information sources and the relationships types captured within the inheritance tree, allows for a more precise detection of semantic clones.

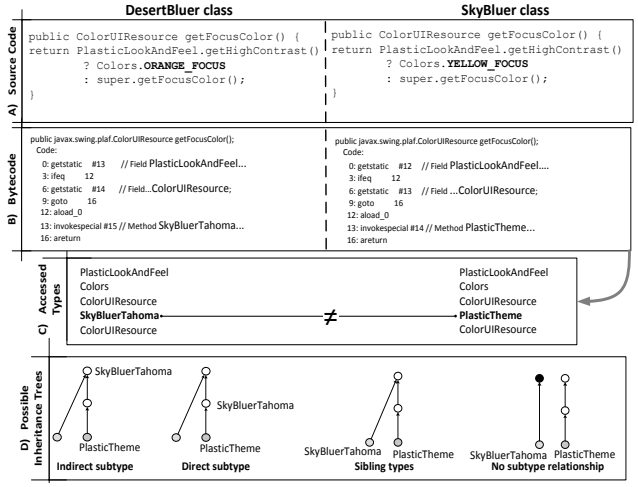

Figure 2. Java source code (A) with corresponding bytecode (B). Section C Highlights the dissimilarities between fragments.

In Fig. 1 and 2, we showed how inheritance tree relationships between entities in code blocks can benefit the clone detection. In order to evaluate the effectiveness of our approach, we analyze the contribution of the inheritance tree information during clone detection. More specifically, what is the effect of the inheritance tree information, via matching (1) sub/super and (2) sibling types within the pattern similarity search process as part of token matching.

For our evaluation we conducted a statistical analysis on four Java code datasets. We measured the potential number of tokens that can be matched (1) since they are identical (the state of the art matching approach used by all major clone

detection approaches), (2) based on their sibling relationship (see Fig. 2-E), and (3) due to their sub/super type. We refer to these last two approaches as inheritance-based matching. Figure 3 shows that in terms of number of tokes matched for the type fingerprints, our inheritance-tree based approach produces a similar number of token matches as the identical token match approaches. Furthermore, the matched tokens reported by the two approaches are not necessarily identical. The observed results confirm not only the applicability of our matching strategy, but also highlight the importance of considering additional information resources (e.g. inheritance trees) as part of clone detection.

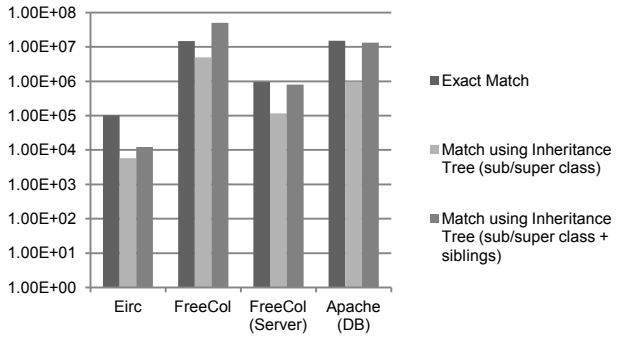

Figure 3. Experimental comparison of number of tokens matched using traditional token and our inheritance-tree based matching approach

III. Common Requirement: Transitive Closure

For our approach to be able to detect semantic clones, we take advantage of two data sources. We combine our novel object oriented inheritance tree representation (Fig. 4 top right) with a traditional code pattern approach that is based on an Abstract Syntax Tree (AST) like representation of the source code (Fig. 4 top left). Both data sources can be represented internally using Tree data structure.

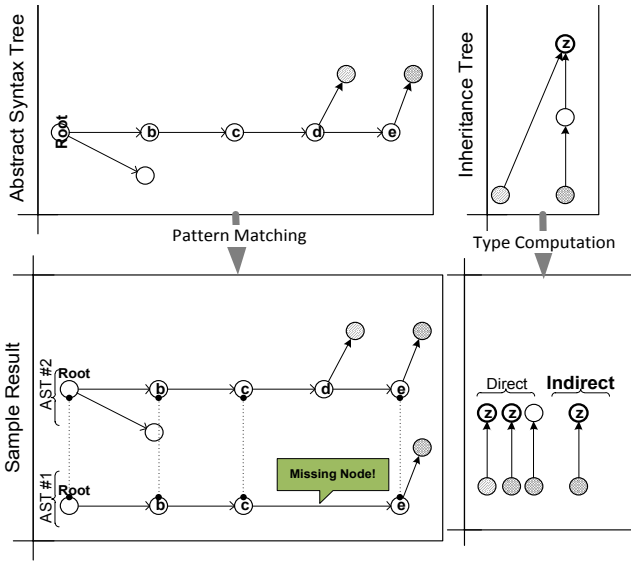

Figure 4. Available information families and their required processing

A. Our Clone Detection Schema

In our approach, we use two types of processing for the clone detection, which are (1) *Type resolution* and (2) *pattern matching*. As part of the *Type resolution,* all classes are identified to which a token belongs. For example in Fig. 4 top right tree, using our type resolution approach, type Z is assigned to both end nodes (Fig. 4 bottom right box). As part of our type resolution approach, types are also assigned if there is no direct subclass link within a given inheritance tree. In this case, our type resolution approach will infer indirect links for the type assignments. We also consider *pattern matching*, to find exact or similar sequences of code fragments (i.e., type-1, 2, and gapped type-3 clones). For example in the AST #1 and #2 (Fig. 4 bottom left) most pattern matching approaches will detect this similar (gapped) ASTs as clones, although they are not exactly identical (since they constitute a type-3 clone case).

B. Reachability

So far we have introduced the two data sources relevant for our approach, namely (1) Abstract Syntax and (2) Inheritance Tree and discussed the processing that is required on these two data sources to detect clones. In what follows, we argue that both are based on a common problem known as the *reachability* problem.

For our *type resolution*, we are interested in determining all types to which either a Java variable or class belongs. Having all variables and classes presented as nodes within the tree, inheritance relationships can then be represented as directed edges between nodes. Given this representation, one can now transform the type resolution to a reachability problem to address the question: What are all *reachable* nodes from each sub-node in the tree? *Reachability* has been previously defined in graph theory [10] as cases when there exist either *direct* or *indirect* paths between two nodes in a tree. An indirect path refers to the situation where there are some middle nodes in between a source and the destination nodes on the detected path.

In *pattern matching*, we have two ASTs - the querying AST tree (e.g., AST #1 in Fig. 4 left bottom box) and the matching tree (e.g., AST #2 in Fig. 4 left bottom box). The objective of pattern matching is to determine whether the: (1) matching tree is exactly similar to the querying tree or (2) if the matching tree is almost similar, except for extra nodes and edges. Fig. 4 left bottom box illustrates an example for such a similarity matching scenario, where the matching tree (i.e., AST #2) has some extra nodes (e.g., node d). In this case, the pattern matching process compares corresponding consecutively nodes in the two trees starting from the root. For example in Fig. 4 bottom left box, it starts by comparing the roots. The search process continues with the next node (i.e., node b) in the querying tree (i.e., AST #1) to detect if there is a node b *reachable* from its root. This pattern matching process continues recursively until the complete tree was searched. This type of pattern matching, can be re-formulated as a reachability problem, to detect whether there are two matching direct or indirect (e.g., gapped type-3 clones) paths between two given nodes.

C. Directed Acyclic Graph

Up to now, we discussed our two core processing steps to identify semantic clones. However, to perform the actual semantic clone detection, the two data sources have to be merged (combining AST and inheritance tree) and aligned (common nodes in both trees) in a unified representation. This merging task is straight forward (shown in Fig. 5) and results in an internal Directed Acyclic Graph (DAG) data structure.

DAG correspond to a set of vertices V and a set of directed edges (i.e., relation) R. For example, $v_1 \dot{R}' v_2$ denotes a concrete link between two nodes known as $v_1$and v_2. It is also possible to have a parallel path such as $v_1 \ddot{R} v_2$. Note that, since it is an acyclic graph there must not be any direct or indirect path as $v_2 \dot{R} v_1$ and $v_2 \ddot{R} v_1$ [10]. Compared to a *tree* data structure, the DAG provides more expressiveness and requires a more complex algorithm to solve the *reachability problem* due to possible parallel paths in the DAG.

D. Solving the Reachability through Transitive Closure

In what follows we discuss how the reachability problem on DAG can be resolved using *transitive closure*. Transitive closure can be defined as follows. If R^+ denotes a transitive relation and $\{aR^+b,\ bR^+c\}$ is the corpus, then constructing transitive closure introduces a new edge to the corpus which is aR^+c. In order to apply transitive closure on the DAG, related edges in the graph have to me marked as *transitive relation*.

Several algorithms [10, 11, 12] have been proposed to compute transitive closure for general graphs and DAG. These algorithms are all similar to routing algorithms such as the one by Dijkstra. Regardless of the algorithm, computational complexity of transitive closure computation remains high (e.g., $O(|V|^3)$) and is non-deterministic [13]. The overall performance of the algorithms depends on the data characteristics (e.g., number of nodes, graph density, centrality etc.) and the computing platform (e.g. disk-based) [14]. Although transitive closure can be considered a solution to our reachability problem for DAG, its complexity and therefore scalability remains a challenge.

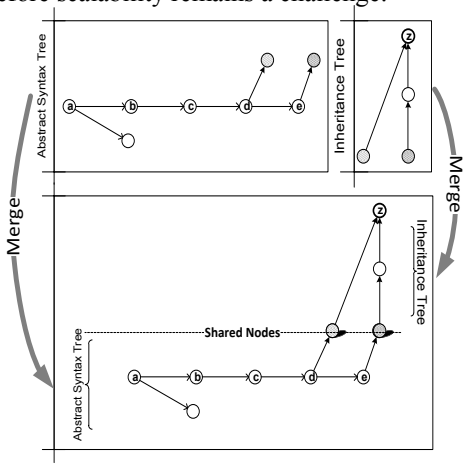

Figure 5. DAG is the common data structure for internal representation

E. Transitivity on DAG via Semantic Web-based Querying and Reasoning

The Semantic Web has emerged as a potential solution for addressing the ambiguity of data on the Internet by making Web content machine processable. It allows knowledge to be formally represented using logics such as description logic (DL). Semantic Web reasoners can infer logical consequences from asserted DL statements. Description Logic (DL) is a family of logic based Knowledge Representation formalisms that can be used to represent the knowledge of an application domain in a structured and formally well-understood way. DL can describe domain in terms of concepts (classes), roles (properties, relationships) and individuals. Today description logic has become a cornerstone of the Semantic Web for its use in the design of ontologies. The Web Ontology Language (OWL) is one of the basic knowledge representation languages on the basis of DL for writing ontologies. It can be considered as a fundamental technology providing a foundation for Semantic Web.

Based on its theoretical foundations, several data modeling languages for graphs (e.g. DAG) and querying languages are proposed, implemented and standardized. OWL is the primary modeling language which supports up to First-Order logic.

SPARQL is its graph-based query language which is compatible with OWL. Similar to other data storage and manipulation systems (e.g. Relational database and Datalog), there are both commercial and free storage and query-endpoints which supports OWL and SPARQL. They support all of OWL and SPARQL features which one of them is transitive closure computation on DAG. Semantic Web reasoners have been optimized for both in-memory and disk-based scalable transitive closure computation during the past decade (e.g., [14]). Moreover, transitive closure computation has been always a challenging requirement to achieve in practice [15]. Therefore our proposed solution is designed based on Semantic Web infrastructure.

IV. Our Semantic Web-based Approach Overview

In this position paper, we discuss how the inclusion of secondary data sources (e.g. inheritance tree) can facilitate and benefit clone detection. Our clone detection approach is based on the use of Semantic Web-based reasoning, which supports transitive closure computing on graph data structures. We chose Semantic Web as the underlying technology since its infrastructure is mature enough both in theory and practice. Figure 6 not only shows the architecture of our approach but also how it is applied for our running example (Fig. 1). We created a few SPARQL queries for different types of clones and executed them against our unified graph knowledge base that includes both inheritance and pattern trees. Using a reasoner-enabled query engine we can then match semantic clones, similar to Fig. 1 and 2. The answer (the top section) in Fig. 6 example is a type-3 fragment, which has not only pattern dissimilarity (i.e. additional for loop) but also explicit type dissimilarity which is matched using inheritance tree (i.e. radiobottom).

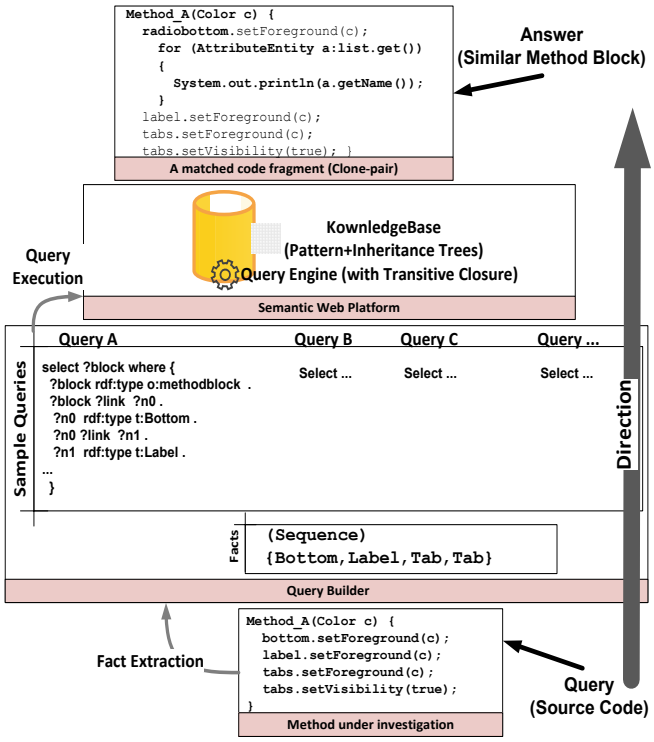

Figure 6. Architecture overview. We also use the sample to show how the internal process works (from bottom to top).

V. Conclusion

The detection of semantic clones can be seen as a complimentary approach to traditional clone detection techniques such as CCFinder [5]. Semantic clone detection supports different application contexts, when semantic similarities plays an important role (e.g. refactoring, restructuring) of source code.

In this paper, we introduced a novel approach to detect semantic clones in source code, which takes advantage of both OO inheritance trees and code patterns as major information sources. Given these two information resources, also different types of processing are required. (1) *Type resolution* and (2) *pattern matching*. Pattern matching identifies exact or similar code fragment sequences and must survive from even *extreme dissimilarities* due to gaps, repetitions, and sliding. However, to support both types of processing, transitive closure computation on DAGs is required. An elegant solution is to use Semantic Web enabled reasoning with its matured enabling infrastructures, as well as its sound theoretical foundation. Therefore, we designed a Semantic Web-based approach to detect clones not only based on pattern but also on type similarities. Our early analysis shows that inheritance tree and token pattern matching can provide similar contributions towards the clone detection. However, further empirical analysis of our approach for different project domains and programming languages is required to validate its efficiency and applicability. Moreover, we are planning to implement our approach as part of our SeCold [16] (secold.org) Linked Data repository and infrastructure. This implementation will allow us to investigate the scalability and applicability of our approach when analyzing very large-scale data repositories (SeCold has indexed so far about 18,000 projects). Furthermore, this integration with SeCold will also facilitate a bi-directional synergy, among the Semantic Web and the source code analysis community, with the Semantic Web community being able to provide enabling technologies for many core challenges in the source code analysis domain. At the other end of the spectrum, the source code community can provide the Semantic Web community with new domain specific application context, datasets and modeling challenges.

References

[1] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programming, vol. 74, no. 7, pp. 470-495, 2009.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Transactions on Software Engineering, , vol. 33, no. 9, pp. 577–591, 2007.

[3] B. S. Baker and U. Manber, "Deducing similarities in Java source from bytecodes," USENIX Annual Technical Conference. 1998.

[4] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," Working Conference on Reverse Engineering, pp. 253-262, 2006.

[5] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654–670, 2002.

[6] Y. Higo and S. Kusumoto, "Enhancing Quality of Code Clone Detection with Program Dependency Graph," Working Conference on Reverse Engineering, pp. 315-316, 2009.

[7] E. Merlo, G. Antoniol, M. D. Penta, and V. F. Rollo, "Linear Complexity Object – Oriented Similarity for Clone Detection and Software Evolution Analyses," IEEE International Conference on Software Maintenance, pp. 0-4, 2004.

[8] M. Tokunaga, N. Yoshida, K. Yoshioka, M. Matsushita, K. Inoue "Towards Collection of Refactoring Patterns Based on Code Clone Classification," Asian Conference on Pattern Languages of Programs, 2011.

[9] M. Fowler, "Refactoring: Improving the Design of Existing Code," Lecture notes in computer science, 2418, pp. 256, 2002.

[10] Y. Ioannidis, R. Ramakrishnan, and L. Winger, "Transitive closure algorithms based on graph traversal," ACM Trans. Database Syst. 18, 3, 512-576, 1993.

[11] B. Roy, "Transitivité et connexité," C. R. Acad. Sci. Paris 249, 216-218, 1959.

[12] S. Warshall, "A Theorem on Boolean Matrices." J. ACM 9, 11-12, 1962.

[13] S. Dar and R. Ramakrishnan, "A performance study of transitive closure algorithms," ACM SIGMOD International Conference on Management of Data, 1994.

[14] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal, "OWL reasoning with WebPIE: calculating the closure of 100 billion triples," The Semantic Web: Research and Applications, pp. 213–227, 2010.

[15] D. Beyer, A. Noack, and C. Lewerentz, "Efficient relational calculation for software analysis," IEEE Transactions on Software, vol. 31, no. 2, pp. 137-149, 2005.

[16] I. Keivanloo, C. Forbes, and J. Rilling, "Towards sharing source code facts using linked data," ICSE International Workshop on Search-driven Development: Users, Infrastructure, Tools, and Evaluation, pp. 25-28, 2011.