Kato: A Plagiarism-Detection Tool for Answer-Set Programs*

Johannes Oetsch, Martin Schwengerer, and Hans Tompits

Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,schwengerer,tompits}@kr.tuwien.ac.at

Abstract. We present the tool Kato which is, to the best of our knowledge, the first tool for plagiarism detection that is directly tailored for answer-set programming (ASP). Kato aims at finding similarities between (segments of) logic programs to help detecting cases of plagiarism. Currently, the tool is realised for DLV programs but it is designed to handle various logic-programming syntax versions. We review basic features and the underlying methodology of the tool.

1 Background

With the rise of the Internet and its easy access of information, plagiarism is a growing problem not only in academia but also in science and technology in general. In software development, plagiarism involves copying (parts of) a program without revealing the source where it was copied from. The relevance of plagiarism detection for conventional program development is well acknowledged [1]—it is not only motivated by an academic setting to prevent students from violating good academic standards, but also by the urge to retain the control of program code in industrial software development projects.

We are concerned with plagiarism detection in the context of answer-set programming (ASP) [2]. In particular, we deal with disjunctive logic programs under the answer-set semantics [3]. Answer-set programming is characterised by the feature that problems are encoded in terms of theories such that the solutions of a problem instance correspond to certain models (the "answer sets") of the corresponding theory. It differs from imperative languages like C++ or Java (and also to some extent from Prolog) because of its genuine declarative nature: a logic program is a specification rather than an instruction of how to solve a problem; the order of the rules and the order of the literals within the heads and bodies of the rules have no effect on the semantics of a program. Hence, someone who copies code has other means to disguise the deed.

For conventional programming languages, sophisticated tools for plagiarism detection exist, like, e.g., YAP3 [4], Sim [5], JPlag [6], XPlag [7], and others [8]. However, most techniques are not adequate for ASP. The reason is the declarative nature of ASP as well as the lack of a control flow. Especially the fact that the order of statements (and of the literals of a statement) is not relevant for a program causes that existing

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

techniques are unsuitable in general. As well, many commonly used tools work with a tokenisation: the source code is translated into a token string where code strings are replaced by generic tokens. For instance, a tokeniser could replace each concrete number by the abstract token <VALUE>. The resulting token strings are used for the further comparisons by searching for common substrings. However, the structure of a DLV program is rather homogeneous—there are not many built-in predicates—which makes this technique unsuitable for detecting copies.

The need for tools for plagiarism detection in ASP is clearly motivated by the growing application in academia and industry, but our primary interest to have such a tool is to apply it in the context of a laboratory course at our university. We thus developed the tool Kato which, to the best of our knowledge, is the first system for plagiarism detection that is directly tailored for ASP.[1] Kato aims at finding similarities between (segments of) logic programs to help detecting cases of plagiarism. Currently, the tool is realised for DLV programs but it is designed to handle various logic-programming syntax versions as well.[2] In what follows, we review the basic features of Kato and outline its underlying methodology.

2 Features and Basic Methodology of Kato

Kato was developed to find suspicious pairs of programs stemming from student assignments in the context of a course on logic programming at our university. Hence, the tool can perform pairwise similarity tests on a rather large set of relatively small programs. In what follows, we provide basic information concerning the implemented features of Kato and how they are realised.

Figure 1 shows the basic working steps needed to perform a test run, which can be divided into three major phases: First, the programs are parsed and normalised in a preprocessing step. Then, test specific preparations are applied. Finally, the programs are compared.

Following a hybrid approach, Kato performs four kinds of comparison tests, realising different layers of granularity: (i) a comparison of the program comments via the longest common subsequence (LCS) metric (see the work of Bergroth et al. [9] for an overview), (ii) an LCS test on the whole program, (iii) a fingerprint test, and (iv) a structure test. We recall that the LCS of two strings is the longest set of identical symbols in both strings with the same order. Hence, the LCS metric tolerates injected non-matching objects. Note that (i) and (ii) are language independent while (iii) and (iv) need to be adapted for different language dialects. All of these tests, outlined in more detail below, compare files pairwise and return a similarity value between 0 (no similarities) and 1 (perfect match).

LCS-Comment Tests. It is surprising what little effort some people spend to mask copied comments. This test reveals similarities between program comments when interpreted as simple strings via the LCS metric.

[1] The name of the tool derives, with all due acknowledgments, from Inspector Clouseau's loyal servant and side-kick, Kato.

[2] See http://www.dlvsystem.com/ for details about DLV.

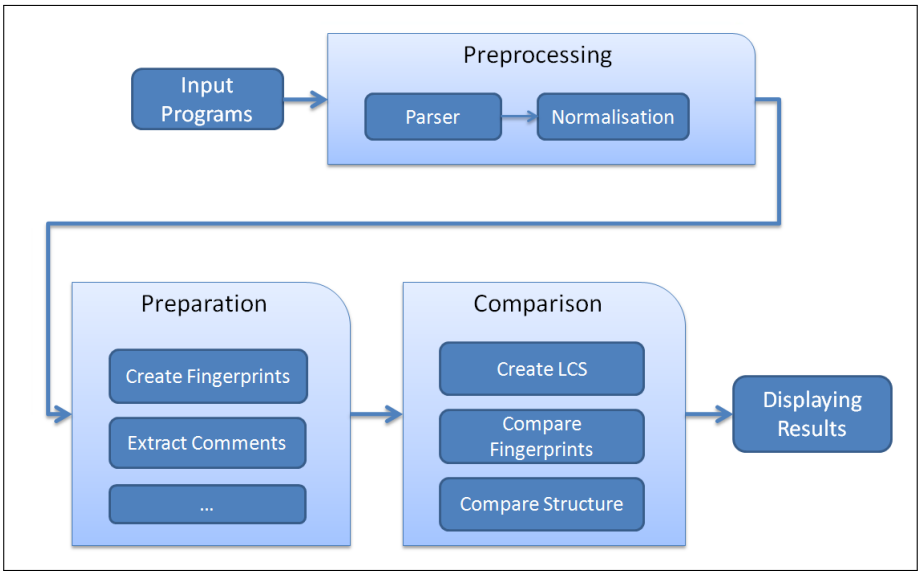

Fig. 1. Overview over a test run

LCS-Program Tests. Similar to the LCS-comment test, the whole programs are interpreted as two strings which are then tested for their longest common subsequence. This test represents an efficient method to detect cases of plagiarism where not much time has been spent to camouflage the plagiarism or parts of it.

Fingerprint Tests. A fingerprint of a program is a collection of relevant statistical data like hash-codes, the number of rules, the number of predicates, the number of constants, program size, and so on. After fingerprints of all programs are generated, they are compared pairwise. This gives a simple yet convenient way to collect further evidence for plagiarism.

Structure Tests. This kind of tests gives, by taking the structure of the programs into account, the most significant information in general. The central similarity function underlying the structure tests is defined as follows: Let $lit_H(r)$ be the multiset of all literals occurring in the head of a given rule r and $lit_B(r)$ the multiset of all literals occurring in the body of r. Then, for two rules r_1 and r_2, the *rule similarity*, $\sigma(r_1, r_2)$, is defined as

$$\sigma(r_1, r_2) = \frac{|lit_H(r_1) \cap lit_H(r_2)| + |lit_B(r_1) \cap lit_B(r_2)|}{\max(|lit_H(r_1)| + |lit_B(r_1)|, |lit_H(r_2)| + |lit_B(r_2)|)}.$$

Furthermore, for two programs P_1 and P_2 (interpreted as multisets of rules), the *similarity*, $\mathcal{S}(P_1, P_2)$, is given by

$$\mathcal{S}(P_1, P_2) = \frac{\sum_{r \in P_1} \max(\sigma(r, r') : r' \in P_2)}{|P_1|}.$$

Note that $\mathcal{S}$ is not symmetrical in its arguments. For any two programs P_1 and P_2, $\mathcal{S}(P_1, P_2)$ is mapped to a value between 0 and 1 which, roughly speaking, expresses to which extent P_1 is subsumed by P_2 by similar rules.

By definition, $\mathcal{S}$ thwarts disguising strategies like permuting rules or literals within rules. However, a more advanced plagiarist could also uniformly rename variables within rules or rename some auxiliary predicates. Therefore, our similarity test comes with different levels of abstraction to counter these malicious efforts. Such renaming is handled by finding and applying suitable substitution functions. Without going into details, the problem of finding such functions is closely related to the homomorphism problem for relational structures which is known to be NP-complete. To circumvent the high complexity, we use an efficient greedy heuristic to obtain our substitutions.

To make the similarity function sensitive to common rule patterns, we also implemented a context dependent extension: A *global occurrence table* gives additional information how specific two rules are. The main idea is that rare rules yield better evidence for a copy than common ones. Therefore, `Kato` collects and counts all rules in the considered corpus of programs and stores this information in an occurrence table. During the comparison, the rule similarity is then weighted depending on the frequency of the involved rules.

3 Further Information and Discussion

The tool is entirely developed in Java (version 6.0). The results of the program comparisons are displayed in tabular form with features like sorting and filtering. For the structure tests, the tool shows program pairs and highlights similar rules. Currently, `Kato` is designed for `DLV`'s language dialect but it can be easily extended to other dialects—it is planned to consider standard Prolog as well. `Kato` was successfully applied in a logic programming course at our university; all cases of plagiarism detected by the supervisors showed high similarities, and even further cases of plagiarism could be detected.

Detailed empirical analyses in terms of *precision* and *recall*, as well as comparisons of our approach with existing tools for plagiarism detection, are left for future work. A further topic for future work is to develop means to visualise the comparison results, e.g., to spot clusters of cooperating plagiarists more easily.

Another interesting aspect of `Kato` is a possible use as a software engineering tool: If a team is working on a program, different versions will emerge. Then, the question about the actual differences between two versions is immanent. `Kato` can be adapted to answer such questions.

Additional information about the tool, and how to obtain it, can also be found at

`http://www.kr.tuwien.ac.at/research/systems/kato`.

References

1. Clough, P.: Plagiarism in natural and programming languages: An overview of current tools and technologies. Technical Report CS-00-05, Department of Computer Science, University of Sheffield, UK (2000)

2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge, England (2003)
3. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
4. Verco, K.L., Wise, M.J.: YAP3 : Improved detection of similarities in computer program and other texts. In Klee, K.J., ed.: Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education, New York, ACM Press (1996) 130–134
5. Gitchell, D., Tran, N.: Sim: A utility for detecting similarity in computer programs. In: Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education, ACM Press (1999) 266–270
6. Prechelt, L., Malpohl, G., Philippsen, M.: JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Fakultät für Informatik Universität Karlsruhe, Germany. (2000)
7. Arwin, C., Tahaghoghi, S.M.M.: Plagiarism detection across programming languages. In: Proceedings of the Twenty-Ninth Australasian Computer Science Conference (ACSC 2006). Volume 48 of CRPIT. Hobart, Australia, ACS (2006) 277–286
8. Jones, E.L.: Metrics based plagiarism monitoring. In: Proceedings of the Sixth Annual CCSC Northeastern Conference. (2001) 1–8
9. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: SPIRE. (2000) 39–48