TLP **10** (4–6): 759–775, 2010. © Cambridge University Press 2010
doi:10.1017/S1471068410000402

The system Kato: Detecting cases of plagiarism for answer-set programs

JOHANNES OETSCH, JÖRG PÜHRER, MARTIN SCHWENGERER
and HANS TOMPITS*

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

submitted 8 February 2010; revised 11 April 2010; accepted 21 May 2010

Abstract

Plagiarism detection is a growing need among educational institutions and solutions for different purposes exist. An important field in this direction is detecting cases of *source-code plagiarism*. In this paper, we present the tool Kato for supporting the detection of this kind of plagiarism in the area of answer-set programming (ASP). Currently, the tool is implemented for DLV programs but it is designed to handle other logic-programming dialects as well. We review the basic features of Kato, introduce its theoretical underpinnings, and discuss an application of Kato for plagiarism detection in the context of courses on logic programming at the Vienna University of Technology.

KEYWORDS: answer-set programming, program analysis, plagiarism detection

1 Introduction

With the rise of the Internet and its easy access to 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 (Clough 2000)—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.

In this paper, we deal with plagiarism detection in the context of answer-set programming (ASP), an important logic-programming formalism for declarative problem solving. We discuss the main features and formal underpinnings of the system Kato, a tool developed for supporting the detection of source-code plagiarism for answer-set programs[1]. Currently, the tool is realised for logic programs adhering to the syntax supported by the well-known ASP solver DLV, but it is designed to handle other syntactic dialects as well[2].

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

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

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

Programming in ASP is characterised by the feature that solutions of encoded problems are determined by certain models (the "answer sets") of the corresponding programs. Moreover, the declarative nature of logic programs also marks a notable difference to imperative languages like C++ or Java: a logic program is an executable specification rather than an instruction on how to solve a problem; the order of the rules and the order of the literals within the heads and bodies of the rules do not affect the semantics of a program. As well, logic programs are devoid of any control flow. Hence, as far as plagiarism detection is concerned, someone who copies code has other means to disguise the deed. Consequently, plagiarism detection tools for imperative programming languages, like, e.g., `YAP3` (Verco and Wise 1996b), `Sim` (Gitchell and Tran 1999), `JPlag` (Prechelt *et al.* 2000), `XPlag` (Arwin and Tahaghoghi 2006), and others (Jones 2001), are not adequate for ASP and thus dedicated methods are needed.

The need for tools for plagiarism detection in ASP can be motivated by the growing application in academia and industry, but our primary interest to have such a tool is to use it in connection with laboratory courses on logic programming and knowledge-based systems at our university each involving more than 100 students. Since a manual inspection of student programs resulting from such a large body of participants is rather time consuming, we developed the tool `Kato` to support our grading efforts. `Kato` implements program-based features, like string-based comparisons of program comments, string-based comparisons of entire program sources, fingerprint tests, and structure-based comparisons of programs, as well as a context-dependent confidence measure regarding suspicious code.

A plagiarism detection system applicable in the realm of declarative programming is `Match`, implementing front-ends for Prolog and SML and following a general approach for plagiarism detection (Lukácsy and Szeredi 2005, 2009). For Prolog, `Match` basically compares the program structure given by a program's call graph. Although this approach can be readily adopted for ASP, the call graph of a program as discrimination criterion is somewhat too weak for the particular ASP setting we are considering. A more detailed discussion of the `Match` approach and its relation to ASP and our setting, as well as a discussion of a related idea to compute similarities between predicate definitions due to Serebrenik and Vanhoof (2007), is given in Section 3.

2 Preliminaries on answer-set programming

We are concerned with *disjunctive logic programs under the answer-set semantics* (Gelfond and Lifschitz 1991), a widely used realisation of the ASP paradigm, consisting of rules of form

$$a_1 \vee \cdots \vee a_k \leftarrow a_{k+1}, \ldots, a_l, \text{not } a_{l+1}, \ldots, \text{not } a_m\,, \tag{1}$$

where all a_i are literals, i.e., atoms possibly preceded by the symbol for classical negation $\neg$, over a function-free first-order language and "not" denotes default negation. The set of all rules of form (1) is denoted by $\mathscr{R}$. Given a rule r of form (1), we define the *head* of r as $H(r) = \{a_1, \ldots, a_k\}$, the *positive body* as $B^+(r) = \{a_{k+1}, \ldots, a_l\}$, and the *negative body* as $B^-(r) = \{a_{l+1}, \ldots, a_m\}$. We call r a *constraint* if $H(r) = \emptyset$ and $B^+(r) \cup B^-(r) \neq \emptyset$.

The *answer sets* of a program are defined by a fixed-point construction involving the *Gelfond-Lifschitz reduct* of a program (Gelfond and Lifschitz 1991); we omit details

```
     ⎧ s(C,o1,X) :- combi(C), s(C,i1,X), not ab(C).
P =  ⎨ t(C,o1,X) :- combi(C), t(C,i1,Y), d(C,i2,Z), s(C,o1,1),
     ⎪              d(C,i3,heat), A = Z * 2, X = Y + A, Y <= 40,
     ⎩              not ab(C).

     ⎧ t(X,o1,Y) :- not ab(X),  d(X,i3,heat), d(X,i2,A),
     ⎪      s(X,o1,1), t(X,i1,Z),
     ⎪      40 >= Z,  Y = Z + B,
Q =  ⎨      B = 2 * A,  c(X).
     ⎪
     ⎪ s(X,o1,Y) :- not ab(X),
     ⎪      s(X,i1,Y), c(X).
     ⎩ c(X) :- combi(X).
```

Fig. 1. Example of a program P and a disguised copy Q.

because we are interested in syntactic issues only. Important to note, however, is that the order of literals in a rule, as well as the order of rules in a program, is not relevant for the semantics.

Besides the basic syntax of rules as described above, the language of the system DLV, which Kato is able to process, contains also *built-in predicates* for comparisons and basic integer arithmetics: =, +, *, and !=. For !=, the alternative syntactic form <> is supported. Further built-in predicates for comparisons that are supported by DLV are <=, <, >=, and >. Moreover, DLV supports *aggregate functions* to express certain properties of sets, like #sum or #count, and *weak constraints* that allow to represent optimisation problems.

The two programs in Figure 1 illustrate, on the one hand, the syntax of DLV and, on the other hand, an attempt to disguise a copied program in ASP: Q is a modified version of P resulting from the latter by a combination of permuting and renaming expressions in P, rewriting arithmetic expressions to equivalent ones, and adding the auxiliary predicate c. We will use P and Q as running example in the remainder of this paper.

3 Background and related work on plagiarism detection

3.1 Text and program plagiarism

A multitude of different approaches towards plagiarism detection for documents exists in the literature (Maurer *et al*. 2006). The two main application areas of text-based plagiarism detection are *literature* (or *plain text*) *plagiarism* and *program* (or *source-code*) *plagiarism*. Interestingly, program plagiarism detection has a longer tradition than literature plagiarism, dating back to the 1970s (Ottenstein 1976), whereas extensive research on literature plagiarism detection started in the late 1990s (Austin and Brown 1999; Farringdon 1996). The reason for this is because, on the one hand, computer programs are well structured and therefore easier to analyse and to compare than natural language and, on the other hand, the interest in literature plagiarism is related to the boost of plagiarism that came with the increased availability of information on the Internet.

In both cases, the key task is determining whether a given document is similar to an original work. It is not sufficient for plagiarism detection systems to check for exact copies only—they need to account for different camouflage strategies. To this end, they employ diverse techniques that provide a similarity measure with respect to some metric, some of which are tailored towards the language of the considered document whereas others are language independent. Methods that require knowledge of the language include the comparison of writing style or frequency of spelling errors in literature plagiarism or checking similarities in the control flow in program plagiarism (when imperative languages are considered).

Among language independent methods are basic string similarity checks such as *greedy string tiling* (Wise 1993) or *longest common subsequence* (*LCS*) *tests* (Bergroth *et al.* 2000) which are implemented in many plagiarism detectors and which can be used for plain text as well as for programming languages.

In program plagiarism detection, two general kinds of similarity tests for programs are distinguished: *fingerprint tests* and *structure tests* (Whale 1990). Fingerprint-based (or *attribute-based*) methods search for copies by comparing characteristic attributes such as the numbers of unique and total operators, which provide the basis for the program similarity metrics by Halstead (1977). Structure tests detect similarities based on the actual content and layout of the compared programs. That most modern tools are mainly based on structure-based techniques (Verco and Wise 1996a; Mozgovoy 2008) suggests that they are more accurate than fingerprint tests in certain applications.

A common pattern for the realisation of structure tests is applying string-similarity tests after a preprocessing step, where the program representation is harmonised with respect to certain criteria. An important preprocessing technique in this respect is *tokenisation*, where the source code is translated into a token string such that certain code strings are replaced by generic tokens. The resulting token strings are then used for 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 rather unsuitable for detecting copies.

In general, plagiarism detection tools can be classified into systems that operate within a given *corpus*, i.e., a collection of documents to check for plagiarism, and systems that check documents also with external sources such as the World Wide Web or textbooks (Lancaster *et al.* 2005). Plagiarism detection for programming assignments is typically done intra-corpal, as the problems to solve are usually very specific such that it is hard for students to find adequate external solutions, thus they copy work from their peers.

3.2 Plagiarism detection for logic programs

We next discuss structure-based approaches suitable for declarative languages and relate them to our ASP setting. To start with, Lukácsy and Szeredi (2005; 2009) introduced a general framework for plagiarism detection that is applicable for both procedural and declarative languages. There, the basic idea is to translate source programs into a suitable formal representation and to apply similarity measures defined on these abstract representations. The system `Match` instantiates this framework for, among others, Prolog as source language. A logic program P is translated into its *call graph*, also known as *predicate-dependency graph*, which is a directed graph where the nodes are the predicate symbols in P, and there is a directed edge between any two nodes a and b whenever P contains a rule with a in its head and b in its body. Then, the similarity between two programs is assessed by computing a graph similarity measure based on graph isomorphism between the corresponding graph representations. The framework also allows to rise the level of abstraction by certain reduction steps on the graph representation.

To use a dependency-graph representation for programs along with graph-theoretic similarity measures is certainly an elegant way to counter many common plagiarism covering tricks. Changing names of identifiers and variables, changing the arity of predicates by introducing dummy parameters, or reordering rules and predicates in rule bodies are countered simply by the abstraction due to the graph translation. Other tricks like adding useless rules or putting some auxiliary or intermediate definitions into a program are reflected by respective structural properties of the graph representations.

In principle, the framework implemented by `Match` can be adapted for answer-set programs as well. However, we follow an alternative approach due to certain particularities of ASP and the setting we are interested in. More specifically, ASP encodings tend to be quite concise in general, and especially the programs we had to check in the context of our courses on logic programming (the primary application area of our tool) consist of few rules only. Moreover, the language and the programs are rather rigorously specified. Hence, the structure of the dependency graph is almost identical for most pairs of programs which disqualifies the form of program abstraction used in `Match` for our purposes. Instead, we address the similarity between two programs at the *rule level*, where rules in one program are matched with similar rules in the other program, and a rather fine-grained control over the abstraction level of this rule-matching procedure is introduced.

A further approach that can be adapted for plagiarism detection for logic programs is one by Serebrenik and Vanhoof (2007) who investigated methods to measure the similarity between two predicate definitions. Interestingly, the initial motivation of this work was to detect and eliminate duplicated code that possibly resulted from refactoring steps. In that approach, two predicate definitions are considered to be similar if they have the same recursive structure modulo renamings and permutations of argument positions. The actual similarity of two predicate definitions is computed, put simply, as the sum of corresponding clause similarities that are calculated recursively for the involved body predicates and reflect the amount of common structure that can be preserved when using suitable generalisations. To ease the comparison of predicates, the authors introduce a fingerprinting technique, where a fingerprint is an abstraction of the recursive structure of a predicate. Thus, this notion of predicate similarity accounts for consistent renamings and permutations and seems to be promising for plagiarism detection for declarative logic-based languages as well.

A possible shortcoming of that approach when used for plagiarism detection is, however, that the notion of similarity between two predicates is rather semantic than syntactic. That is to say, the form of two definitions can be quite different although their similarity is quite high. This is adequate for detecting duplicated code but we aim for a similarity notion that is closer tied to the syntactic form of the two programs that are compared.

4 The System `Kato`: Architecture and basic features

`Kato` was developed to find pairs of programs which are suspicious with respect to plagiarism. The considered programs stem from student assignments from a course on logic programming at the Vienna University of Technology. `Kato` thus can perform pairwise

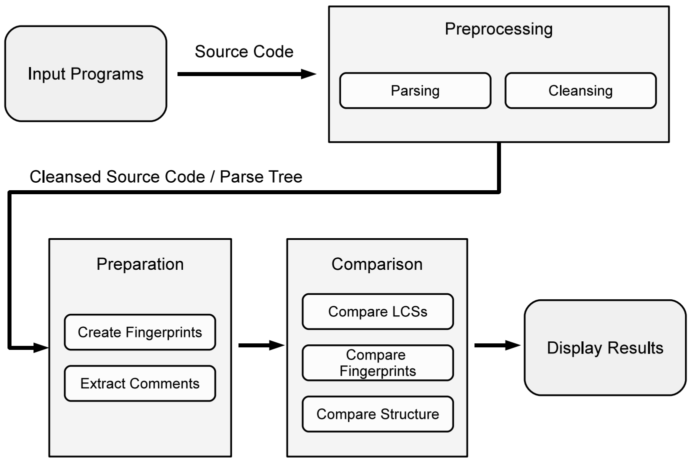


Fig. 2. Overview of how programs are compared in `Kato`.

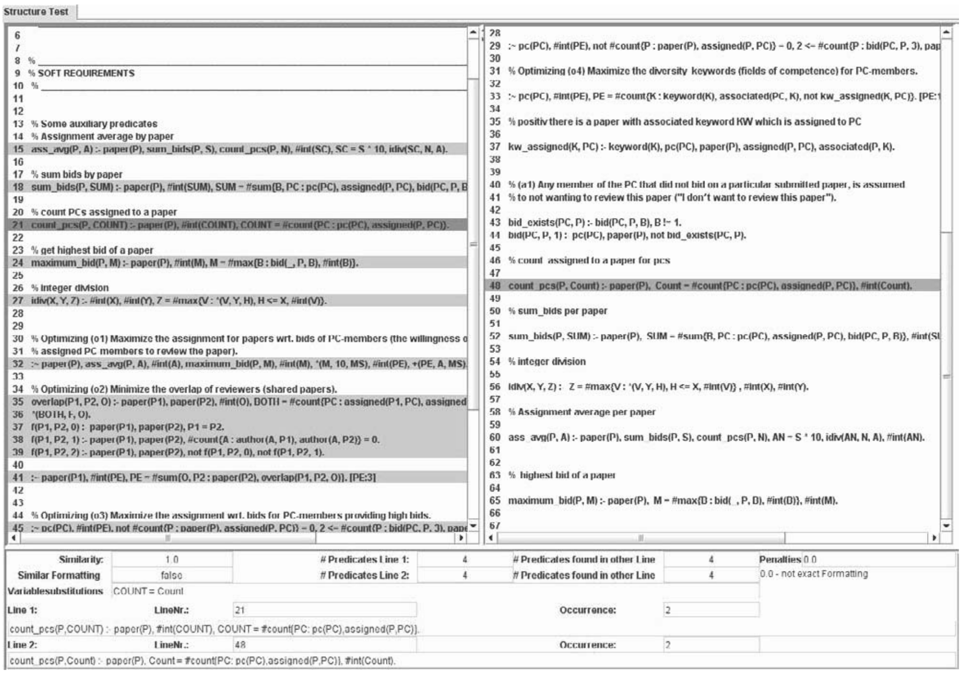
similarity tests on rather large collections of programs. In what follows, we provide basic information concerning the implemented features of `Kato` and how they are realised.

The system was entirely developed in Java (Version 6.0) and provides a graphical user interface for controlling test runs (a test run is a series of pairwise program comparisons on collections of programs or single program pairs). The user can modify the applied tests and configure how the system displays results. The results of a test run can be saved in a file or exported into a MySQL database. Test results are displayed in a table where each line gives the results of the pairwise program comparisons. For results of single tests, `Kato` provides detailed views regarding the computed similarities. Additional information about the tool can be found at

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

Following a hybrid approach, `Kato` performs four kinds of comparison tests, realising different layers of granularity: (i) a string-based comparison of the program comments, (ii) a string-based comparison of entire program sources, (iii) a fingerprint test, and (iv) a structure-based comparison of programs.

For the string-based tests, comparisons based on a longest common subsequence (LCS) metric are applied (Bergroth *et al*. 2000). Long common substrings are usually an indicator for a copy, however a major drawback of this method is its vulnerability with respect to non-matching symbols in a sequence which break the substrings apart (Loose *et al*. 2008). On the other hand, as the LCS of two strings is the longest sequence of symbols appearing in both strings with the same succession, the LCS metric tolerates injected non-matching objects and is more robust in this respect. To illustrate the notion of LCS, consider the two strings "`Kato` is a great tool for plagiarism detection in ASP" and "the system `Kato` is currently applied in a course on logic programming to reveal cases of plagiarism". The LCS of these strings is "`Kato` is a plagiarism". The *LCS similarity* of two strings s and t is the length of the LCS of s and t normalised by the length of s.


Structure Test

```
6
7
8  % ______________________________________________
9  % SOFT REQUIREMENTS
10 % ______________________________________________
11
12
13 % Some auxiliary predicates
14 % Assignment average by paper
15 ass_avg(P, A) :- paper(P), sum_bids(P, S), count_pcs(P, N), #int(SC), SC = S * 10, idiv(SC, N, A).
16
17 % sum bids by paper
18 sum_bids(P, SUM) :- paper(P), #int(SUM), SUM = #sum{B, PC : pc(PC), assigned(P, PC), bid(PC, P, B
19
20 % count PCs assigned to a paper
21 count_pcs(P, COUNT) :- paper(P), #int(COUNT), COUNT = #count{PC : pc(PC), assigned(P, PC)}.
22
23 % get highest bid of a paper
24 maximum_bid(P, M) :- paper(P), #int(M), M = #max{B : bid(_, P, B), #int(B)}.
25
26 % integer division
27 idiv(X, Y, Z) :- #int(X), #int(Y), Z = #max{V : *(V, Y, H), H <= X, #int(V)}.
28
29
30 % Optimizing (o1) Maximize the assignment for papers wrt. bids of PC-members (the willingness o
31 % assigned PC members to review the paper).
32 :~ paper(P), ass_avg(P, A), #int(A), maximum_bid(P, M), #int(M), *(M, 10, MS), #int(PE), +(PE, A, MS).
33
34 % Optimizing (o2) Minimize the overlap of reviewers (shared papers).
35 overlap(P1, P2, O) :- paper(P1), paper(P2), #int(O), BOTH = #count{PC : assigned(P1, PC), assigned
36 *(BOTH, F, O).
37 f(P1, P2, 0) :- paper(P1), paper(P2), P1 = P2.
38 f(P1, P2, 1) :- paper(P1), paper(P2), #count{A : author(A, P1), author(A, P2)} = 0.
39 f(P1, P2, 2) :- paper(P1), paper(P2), not f(P1, P2, 0), not f(P1, P2, 1).
40
41 :- paper(P1), #int(PE), PE = #sum{O, P2 : paper(P2), overlap(P1, P2, O)}. [PE:3]
42
43
44 % Optimizing (o3) Maximize the assignment wrt. bids for PC-members providing high bids.
45 :~ pc(PC), #int(PE), not #count{P : paper(P), assigned(P, PC)} = 0, 2 <= #count{P : bid(PC, P, 3), pape
```

```
28
29 :~ pc(PC), #int(PE), not #count{P : paper(P), assigned(P, PC)} = 0, 2 <= #count{P : bid(PC, P, 3), pap
30
31 % Optimizing (o4) Maximize the diversity keywords (fields of competence) for PC-members.
32
33 :~ pc(PC), #int(PE), PE = #count{K : keyword(K), associated(PC, K), not kw_assigned(K, PC)}. [PE:1
34
35 % positiv there is a paper with associated keyword KW which is assigned to PC
36
37 kw_assigned(K, PC) :- keyword(K), pc(PC), paper(P), assigned(P, PC), associated(P, K).
38
39
40 % (a1) Any member of the PC that did not bid on a particular submitted paper, is assumed
41 % to not wanting to review this paper ("I don't want to review this paper").
42
43 bid_exists(PC, P) :- bid(PC, P, B), B != 1.
44 bid(PC, P, 1) :- pc(PC), paper(P), not bid_exists(PC, P).
45
46 % count assigned to a paper for pcs
47
48 count_pcs(P, Count) :- paper(P), Count = #count{PC : pc(PC), assigned(P, PC)}, #int(Count).
49
50 % sum_bids per paper
51
52 sum_bids(P, SUM) :- paper(P), SUM = #sum{B, PC : pc(PC), assigned(P, PC), bid(PC, P, B)}, #int(SU
53
54 % integer division
55
56 idiv(X, Y, Z) :- Z = #max{V : *(V, Y, H), H <= X, #int(V)} , #int(X), #int(Y).
57
58 % Assignment average per paper
59
60 ass_avg(P, A) :- paper(P), sum_bids(P, S), count_pcs(P, N), AN = S * 10, idiv(AN, N, A), #int(AN).
61
62
63 % highest bid of a paper
64
65 maximum_bid(P, M) :- paper(P), M = #max{B : bid(_, P, B), #int(B)}, #int(M).
66
67
```

Similarity:	1.0	# Predicates Line 1:	4	# Predicates found in other Line	4	Penalties 0.0
Similar Formatting	false	# Predicates Line 2:	4	# Predicates found in other Line	4	0.0 - not exact Formatting
Variablesubstitutions	COUNT = Count					

Line 1: LineNr.: 21 Occurrence: 2

```
count_pcs(P,COUNT) :- paper(P), #int(COUNT), COUNT = #count{PC: pc(PC),assigned(P,PC)}.
```

Line 2: LineNr.: 48 Occurrence: 2

```
count_pcs(P,Count) :- paper(P), Count = #count{PC: pc(PC),assigned(P,PC)}, #int(Count).
```

Fig. 3. A screen-shot showing how `Kato` displays the results of a structure test.

We note that tests (i) and (ii) are language independent whilst (iii) and (iv) need to be adapted for different languages. 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).

Figure 2 shows the basic working steps needed to compare programs in `Kato`. First, the source code of a collection of programs is cleansed, i.e., white spaces are removed, and parsed in a preprocessing step. Then, the cleansed source code, as well as the abstract program representations resulting from the parsing step, are subject to a preparation step where fingerprints are created and comments are extracted. In a comparison phase, LCS comment tests, LCS program tests, fingerprint tests, and structure tests are applied. Finally, the results of the comparisons are presented to the user. More specifically, the results are displayed in tabular form with features like sorting and filtering. For the structure tests, the tool shows program pairs and highlights similar rules; see Figure 3 to get a rough impression.

In what follows, we describe the comparison tests (i)–(iv) in more detail.

LCS comment test. When programs are copied, it is surprisingly quite frequent that program comments are copied as well and usually little effort is spent to mask such copied comments. Consequently, the LCS comment test aims at revealing similar parts in program comments. More specifically, this test reveals similarities between two programs by comparing the (concatenated) comments in the two programs via the LCS similarity measure.

LCS program test. This test is very similar to the LCS comment test but considers entire programs. Thus, this test computes the LCS similarity of the cleansed source code of two programs when interpreted as strings. It turned out that this test is an efficient method to detect cases of plagiarism where not much time has been spent to camouflage the deed. It is especially well-suited to detect perfectly matching sequences in programs which is usually a clear hint for plagiarism.

Fingerprint test. Recall that a fingerprint of a program is a collection of relevant program attributes, i.e., 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 in a preparation step, the fingerprints are compared pairwise. This test is quantified by normalising the number of matching attributes by the total number of considered attributes for each pair of programs. This gives a simple yet convenient way to collect further evidence for plagiarism.

Structure test. This test does not operate on the cleansed program sources but on the program representation that was built in a preceding parsing step. We here rely on a set representation of programs and of rule heads and bodies as well as on a structured representation of literals and term lists. While the LCS comment test, the LCS program test, and the fingerprint test are, more or less, standard techniques to detect plagiarism in general, the structure test is developed specifically for our ASP setting.

The central notion of *program similarity* that underlies the structure test, formally defined in the next section, is designed to thwart disguising strategies like permuting rules or literals within rules. However, a more advanced plagiarist will apply additional camouflage techniques, e.g., uniformly renaming variables in rules or renaming auxiliary predicates in programs. Therefore, our similarity test comes with different levels of abstraction, so-called *camouflage techniques*, to counter such efforts. Without going into details at this point, renaming is handled by finding and applying suitable substitution functions.

To make the similarity function sensitive to common rule patterns, we also implemented a context dependent notion of *confidence* regarding plagiarism: When whole collections of programs are examined for suspicious pairs of programs, a *global occurrence table* gives additional information on 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 which is then used to compute a measure of confidence regarding plagiarism for two programs based on the frequency of common rules in the programs relative to their frequency in the considered corpus.

5 The formal similarity model of the structure test

In this section, we present a syntactic measure of program similarity between logic programs that forms the theoretical basis of the central plagiarism-detection features of `Kato`. Furthermore, we introduce a confidence measure that aims at quantifying the reliability of our similarity measure for detecting actual cases of plagiarism in ASP.

5.1 A syntactic measure for rule and program similarity

Our basic assumption is that a plagiarist camouflages a copy by changing the form of the copied program, i.e., by *program transformations at a syntactic level*. Examples for rather simple camouflage techniques are changing the formatting of the source code, permuting rules of the program, and changing the order of literals within rule heads or bodies. More advanced techniques include uniformly renaming variables within rules, renaming auxiliary atoms within programs, and rewriting arithmetic expressions to equivalent ones. Clearly, all these basic techniques can be combined to build more complex operations.

A key concept for detecting copied programs is thus measuring the similarity between programs. Formally, this boils down to the definition of a suitable similarity function that quantifies the syntactic similarity between two programs with respect to specific combinations of camouflage techniques. To cover different camouflage techniques in a uniform way and to allow for the composition of different techniques, we handle them as functions that map pairs of rules to pairs of rules.

Definition 1
A *camouflage technique*, or *technique*, is a function with domain and co-domain $\mathscr{R} \times \mathscr{R}$. In particular, by the *identity technique*, τ^{id}, we understand the identity function over $\mathscr{R} \times \mathscr{R}$, i.e., τ^{id} satisfies $\tau^{\mathsf{id}}(p) = p$, for each $p \in \mathscr{R} \times \mathscr{R}$.

Obviously, the notion of camouflage technique is closed under functional composition, i.e., for any two techniques τ_1 and τ_2, the composition $\tau_1 \circ \tau_2$ is a technique as well. From an algebraic point of view, the set of techniques and the composition operator form a monoid with τ^{id} as its identity element. Hence, the above definition allows to express more complex combinations of different techniques, as used, e.g., to disguise program Q in Figure 1, by a composition of less complex techniques.

We next introduce our central similarity measures. Roughly speaking, we define the similarity between two single program rules r and s as the number of common head and body literals of r and s normalised by the total number of literals in r.

Definition 2
For any two rules $r, s \in \mathscr{R}$, the *rule similarity between r and s* is given by

$$\sigma(r,s) = \frac{|H(r) \cap H(s)| + |B^{+}(r) \cap B^{+}(s)| + |B^{-}(r) \cap B^{-}(s)|}{|H(r)| + |B^{+}(r)| + |B^{-}(r)|}.$$

For example, consider the two rules

r=`p(X) :- r(X), s(X,Z), not t(c)` and
s=`p(X) v q(Y) :- not t(c), s(X,Z), r(Y).`

According to the above definition, $\sigma(r,s) = \frac{3}{4}$.

The notion of rule similarity extends to entire programs as follows:

Definition 3
Given two programs P_1 and P_2 together with a camouflage technique τ, the *program similarity between P_1 and P_2 with respect to τ* is given by

$$\mathscr{S}_{\tau}(P_1,P_2) = \frac{\sum_{r \in P_1} \max\left\{\sigma\big(\tau(r,r')\big) \mid r' \in P_2\right\}}{|P_1|}.$$

Clearly, for any program P_1 and P_2 and for any technique τ, it holds that $\mathscr{S}_\tau(P_1, P_2) \in [0, 1]$. Note that rule similarity and program similarity are not symmetric in their arguments. Roughly speaking, the significance of this asymmetry is that it allows to express to which extent P_1 is subsumed by P_2 by similar rules with respect to σ and τ. This can be further interpreted as hints concerning to which extent one program resulted from another program by adding or splitting certain rules which can help to assess who copied from whom.

Using the identity technique to the two programs P and Q from Figure 1 to compute its similarity yields $\mathscr{S}_{\tau^{\mathsf{id}}}(P, Q) = \mathscr{S}_{\tau^{\mathsf{id}}}(Q, P) = 0$. Hence, we need more advanced techniques than τ^{id} to get a similarity function of practical use. Accordingly, we next introduce some basic techniques which can serve as building blocks for more complex ones.

5.2 *Basic camouflage techniques*

5.2.1 Variable renaming

The first technique we consider is the uniform renaming of variables within rules which is a simple way to change the form of a rule. Formally, a *variable renaming* for a rule r is a bijection from the set S of variables occurring in r to a set S' consisting of $|S|$ arbitrary variables. For any variable renaming ϑ for some rule r, $r\vartheta$ denotes the result of replacing each occurrence of a variable x in r by $\vartheta(x)$. We assume that $\preceq^{\mathsf{v}}$ is a globally fixed well-ordering on variable renamings that extends to tuples of variable renamings in a lexical way. Informally, the following technique applies variable renamings to a pair of rules that results in a maximal rule similarity. Since such maximal variable renamings are not unique in general, we need $\preceq^{\mathsf{v}}$ to define a proper function.

Definition 4

The camouflage technique τ^{v} is the function assigning to each rule pair $(r, s) \in \mathscr{R} \times \mathscr{R}$ the pair $\tau^{\mathsf{v}}(r, s) = (r\vartheta_r, s\vartheta_s)$, where ϑ_r, ϑ_s are the variable renamings for r and s such that

$$(\vartheta_r, \vartheta_s) = \min_{\preceq^{\mathsf{v}}}\{(\vartheta'_r, \vartheta'_s) \mid \sigma(r\vartheta''_r, s\vartheta''_s) \leqslant \sigma(r\vartheta'_r, s\vartheta'_s), \text{for all variable renamings } \vartheta''_r \text{ and } \vartheta''_s \text{ for } r \text{ and } s \text{ such that } (\vartheta''_r, \vartheta''_s) \neq (\vartheta'_r, \vartheta'_s)\}.$$

For example, consider programs P and Q from Figure 1. Let r be the last rule in P and s the first rule in Q. Rule r contains the variables C, X, Y, Z, and A, and s contains the variables X, Y, Z, A, and B. Variable renamings that yield maximal rule similarity are $\vartheta = \{C \mapsto X, X \mapsto Y, Y \mapsto Z, Z \mapsto A, A \mapsto B\}$ for r and the identity function for s.

Applying ϑ to r results in the rule

```
t(X,o1,Y) :- combi(X), t(X,i1,Z), d(X,i2,A), s(X,o1,1),
             d(X,i3,heat), B = A * 2, Y = Z + B, Z <= 40,
             not ab(X).
```

Assuming a suitable well-ordering, we get $\tau^{\mathsf{v}}(r, s) = (r\vartheta, s)$, and thus $\sigma(\tau^{\mathsf{v}}(r, s)) = \frac{7}{10}$. Note that the similarity between r and s under τ^{id} is 0.

5.2.2 Predicate renaming

Next, we address a technique to deal with renaming efforts at the predicate level. Given a program P, a bijection ϑ from the set S of predicate symbols occurring in P to a set S' consisting of $|S|$ arbitrary predicate symbols is a *predicate renaming* for P if p and $\vartheta(p)$ have the same arity, for each $p \in S$. Let ϑ be a predicate renaming for a program and E a program or a rule. Then, $E\vartheta$ denotes the result of replacing each occurrence of a predicate symbol x in E by $\vartheta(x)$, provided ϑ is defined for x.

Note that predicate renamings are not applied to single rules but to entire programs. To express predicate renamings in terms of camouflage techniques, we introduce a technique that is parameterised by two programs that are used to determine the predicate renaming that is applied[3]. Like for variable renamings, we fix a well-ordering $\preceq^{\mathsf{p}}$ defined on predicate renamings and assume that $\preceq^{\mathsf{p}}$ extends to tuples of predicate renamings in a lexical way.

Definition 5
Given two programs P and Q, the camouflage technique $\tau^{\mathsf{p}}_{P,Q}$ is the function assigning to each rule pair $(r,s) \in \mathscr{R} \times \mathscr{R}$ the pair $\tau^{\mathsf{p}}_{P,Q}(r,s) = (r\vartheta_r, s\vartheta_s)$, where ϑ_r and ϑ_s are the predicate renamings for P and Q such that

$$(\vartheta_r, \vartheta_s) = \min_{\preceq^{\mathsf{p}}}\{(\vartheta'_r, \vartheta'_s) \mid \mathscr{S}_{\tau^v}(P\vartheta''_r, Q\vartheta''_s) \leqslant \mathscr{S}_{\tau^v}(P\vartheta'_r, Q\vartheta'_s), \text{ for all predicate renamings } \vartheta''_r \text{ and } \vartheta''_s \text{ for } P \text{ and } Q \text{ such that } (\vartheta''_r, \vartheta''_s) \neq (\vartheta'_r, \vartheta'_s)\}.$$

Intuitively, the above definition formalises the idea of applying predicate renamings that result in maximal program similarity when also variable renamings are considered for the rule comparisons.

Recall again programs P and Q from Figure 1. Predicate renamings that yield maximal program similarity are the renaming ϑ for P which maps each predicate symbol in P to itself except for `combi`, which is mapped to `c`, and the renaming ϑ' for Q given as the identity function.

Let us consider the technique $\tau = \tau^{\mathsf{p}}_{P,Q} \circ \tau^{\mathsf{v}}$. Then, $\mathscr{S}_\tau(P,Q) = \frac{9}{10} = 0.9$. Observe that $\mathscr{S}_{\tau'}(Q,P) = \frac{18}{30} = 0.6$, for $\tau' = \tau^{\mathsf{p}}_{Q,P} \circ \tau^{\mathsf{v}}$, which illustrates that P is almost entirely subsumed (syntactically) by Q but not vice versa due to the additional third rule in Q.

5.2.3 Canonisation of built-in predicates

The next technique is designed to counter program transformations that exploit that some `DLV` built-in predicates have syntactically different representations. For each built-in predicate, we define a unique *canonical representation*. E.g., the canonical representation of `Z = X + Y` and `+(X,Y,Z)` is `+(X,Y,Z)` (the canonical representations for `*`, `=`, `<>`, `!=`, `<=`, `<`, `>=`, and `>` are omitted for space reasons).

Definition 6
The camouflage technique τ^{c} is the function assigning to each rule pair $(r,s) \in \mathscr{R} \times \mathscr{R}$ the pair $\tau^{\mathsf{c}}(r,s) = (r', s')$, where r' results from r by rewriting all occurrences of built-in predicates in r into their respective canonical forms, and likewise for s' and s.

[3] Formally, this means that we introduce a family of techniques, one technique for each program pair, respectively.

Concerning our running example from Figure 1, if we pool the techniques in our arsenal of camouflage techniques considered so far, we get a program similarity of $\mathscr{S}_{\tau}(P, Q) = \frac{19}{20}$, for $\tau = \tau^{\mathsf{p}}_{P,Q} \circ \tau^{\mathsf{v}} \circ \tau^{\mathsf{c}}$.

5.2.4 *Ordering the arguments of commutative predicates*

For most `DLV` built-in predicates, viz. for `=`, `+`, `*`, `!=`, and `<>`, (some of) their arguments can be commuted. Swapping arguments of such *commutative predicates* is addressed by the next technique.

Definition 7
The camouflage technique τ^{o} is the function assigning to each rule pair $(r, s) \in \mathscr{R} \times \mathscr{R}$ the pair $\tau^{\mathsf{o}}(r, s) = (r', s')$, where r' is the rule that results from r by replacing each commutative predicate p in r with a rewriting of p where all commutable arguments of p are lexically ordered according to their string representations, and s' results from s in an analogous fashion.

For programs P and Q from Figure 1, we eventually obtain $\mathscr{S}_{\tau}(P, Q) = 1$, for $\tau = \tau^{\mathsf{p}}_{P,Q} \circ \tau^{\mathsf{v}} \circ \tau^{\mathsf{c}} \circ \tau^{\mathsf{o}}$, and $\mathscr{S}_{\tau'}(Q, P) = \frac{2}{3} = 0.\dot{6}$, for $\tau' = \tau^{\mathsf{p}}_{Q,P} \circ \tau^{\mathsf{v}} \circ \tau^{\mathsf{c}} \circ \tau^{\mathsf{o}}$.

5.2.5 *Further issues*

Customisation of techniques in `Kato`. Versions of all the basic camouflage techniques discussed so far are implemented in `Kato`. Note, however, that the composition operation for these techniques is not commutative. Thus, different compositions result in different similarity values between programs in general. As well, the user can easily define further techniques from the basic ones, but our current implementation is restricted to compositions subject to the order $\tau^{\mathsf{o}} < \tau^{\mathsf{v}} < \tau^{\mathsf{c}} < \tau^{\mathsf{p}}_{P,Q} < \tau^{\mathsf{id}}$ which turned out to be particularly useful in practice. We plan to extend `Kato` such that this limitation is removed in future versions. Furthermore, `Kato` provides the following hierarchy of predefined techniques:

$$\tau_1 = \tau^{\mathsf{id}},\ \tau_2 = \tau^{\mathsf{v}},\ \tau_3 = \tau^{\mathsf{o}} \circ \tau^{\mathsf{v}} \circ \tau^{\mathsf{c}}, \text{ and } \tau_4 = \tau^{\mathsf{o}} \circ \tau^{\mathsf{v}} \circ \tau^{\mathsf{c}} \circ \tau^{\mathsf{p}}_{P,Q}.$$

Each of these techniques realises a level of abstraction: On the one hand, for two programs P and Q, usually $\mathscr{S}_{\tau_i}(P, Q) \leqslant \mathscr{S}_{\tau_j}(P, Q)$ holds if $i < j$. On the other hand, the significance of a high similarity value relative to τ_i is indirect proportional to i. It often makes sense to consider a sequence of comparisons, starting with τ_1 and ending with τ_4, and to consider similarity and confidence relative to the level i—`Kato` can perform such hierarchical tests, if required. We will discuss an empirical analysis employing these predefined techniques in Section 6.

Complexity aspects. Concerning the inherent complexity of applying a technique to a pair of rules, we note that the variable and predicate renaming techniques tend to be computationally expensive while the other techniques can be applied in linear time with respect to the size of the rules. Without going into details, the problem of finding variable or predicate substitutions is related to the homomorphism problem for relational structures which is known to be NP-complete (when formulated as a decision problem). However, high complexity of the involved reasoning tasks does not turn out to be a bottle-neck of our approach since this worst-case complexity is relative to the size of rules or programs and problem encodings in ASP tend to be quite concise and rarely involve a larger number of rules—this is witnessed, e.g., by the collection of benchmark programs used for the recent ASP solver competition (Denecker *et al.* 2009).

5.3 A corpus-based confidence measure

We now introduce a method how to further evaluate the outcome of a program comparison regarding its potential to actually reveal a case of plagiarism. One important aspect in this regard is the influence of the used camouflage technique τ on the program similarity. In general, we can observe a trade-off between the complexity of τ (in terms of operations on rules) and the significance of a high similarity value for revealing plagiarism. As illustrated, the similarity between programs P and Q from Figure 1 ranges from 0 to 1, depending on the used technique. However, the technique used to obtain a similarity of 1 is rather demanding and presumes that a plagiarist applied more advanced methods to cover a copy. Thus, the significance of the similarity outcome is rather low.

A common and in practice important setting is when we have a collection, or corpus, of programs, and our goal is to detect suspicious pairs of programs within this corpus. Such a setting allows to factor in further information that can increase our confidence that a high program similarity actually indicates a case of plagiarism. Here, the basic idea is to consider the relative frequency of rule occurrences with respect to the considered program collection. If two programs have a high similarity but share only very common rules, our confidence will be accordingly low. Likewise, if two programs contain the same extremely rare rule, our confidence for plagiarism will be rather high. Clearly, the notion of "sameness" of a rule depends on the considered camouflage technique τ. Formally, we introduce a suitable equivalence relation based on τ and consider the induced equivalence class to define the relative frequency of a rule with respect to a program corpus.

Given a camouflage technique τ composed from techniques in $\{\tau^{\mathsf{v}}, \tau^{\mathsf{p}}_{P,Q}, \tau^{\mathsf{c}}, \tau^{\mathsf{o}}\}$, let us define the relation $\sim_\tau \subseteq \mathscr{R} \times \mathscr{R}$ as $r \sim_\tau s$ iff $\sigma(\tau(r,s)) = \sigma(\tau(s,r)) = 1$, for any rule $r, s \in \mathscr{R}$. It can be shown that $\sim_\tau$ is an equivalence relation. For a set $S \subseteq \mathscr{R}$ of rules, a rule $r \in S$, and a relation $\sim_\tau$, let $[r]_\tau$ be the equivalence class of r in S under $\sim_\tau$. The *relative frequency* of r with respect to S and τ is given by

$$f_{S,\tau}(r) = \frac{|[r]_\tau|}{|S|}.$$

The notion of confidence regarding two programs is then defined as follows:

Definition 8

Let $\mathscr{P}$ be a collection of programs and τ a camouflage technique composed from techniques in $\{\tau^{\mathsf{v}}, \tau^{\mathsf{p}}_{P,Q}, \tau^{\mathsf{c}}, \tau^{\mathsf{o}}\}$. Furthermore, let S be the set of all rules occurring in $\mathscr{P}$. Then, for programs $P, Q \in \mathscr{P}$, the *confidence regarding P and Q relative to $\mathscr{P}$ and* τ is given by

$$\mathscr{C}_{\mathscr{P},\tau}(P,Q) = \max\{1 - f_{S,\tau}(r) \mid r \in P \text{ such that there is some } s \in Q \text{ with } r \sim_\tau s\}.$$

6 Application of Kato and empirical analysis

As mentioned earlier, our original motivation for the development of Kato was to have a tool to support detecting cases of plagiarism in the context of several laboratory courses.

The particular setting of the courses was as follows: Students were assigned with exercises and had to solve them by means of ASP. The size of the program parts from the student solutions ranged from a few rules to dozens of rules. Besides evaluating the correctness of the solutions, human supervisors had to check for plagiarism. Since the number of students increased from year to year, with typically over 100 students attending the courses, the task to manually inspect all pairs of programs for plagiarism became too time consuming and laborious. We thus decided in early 2009 to develop Kato, which became operational in the summer term of that year. Kato was used to pre-select pairs of programs if their program similarity was beyond a fixed threshold, and the selected programs were then further manually inspected by the supervisors. Kato had to perform tens of thousands of program comparisons which was feasible within hours. Most importantly, the application of the system yielded dramatical savings in labour time.

To provide a more detailed assessment of Kato, in what follows we give an evaluation of Kato in terms of the well-known notions of *precision* and *recall*. In our setting, precision is the ratio between the number of program pairs that are correctly classified by Kato as copies and the total number of pairs that are classified by Kato as copies (i.e., the number of true positives divided by the sum of true positives and false positives), and recall is the ratio between the number of program pairs that are correctly classified as copies and the total number of pairs that are actual copies (i.e., the number of true positives divided by the sum of true positives and false negatives). The purpose of the evaluation is to show the effects of different combinations of techniques for the structure test on precision and recall.

For the present evaluation, we used data from the logic programming course that took place in 2008 where we had to assess 109 programs from the students. We did not use assignments from the courses after 2008 because these were used as reference data in the development phase of Kato and thus are unsuitable for a fair evaluation as the tool is trained on them. In the considered program collection, we manually identified 26 program pairs suspected for plagiarisms.

Table 1 shows the results of our evaluation of different techniques for the structure test, where we considered the predefined techniques τ_1–τ_4 from above. Recall that τ_1 is the simplest technique while τ_4 is the most complex one. The table is organised as follows. In each of the four main rows, we give the precision and recall results for one technique τ_1–τ_4 as well as the time to run the test. For each technique, we considered different values, ranging from 0.85 to 1, for the similarity threshold used to classify whether a pair of programs counts as a case of plagiarism. For each threshold value, we give the number of program pairs that are classified as copies by Kato, i.e., that showed a program similarity above the threshold value, as well as the number of actual copies, i.e., the number of program pairs from the ones classified as copies by Kato that are indeed cases of plagiarism according to a human supervisor. The last two columns depict the actual precision and recall values.

The evaluation clearly shows the trade-off between precision and recall relative to the abstraction level imposed by the considered camouflage technique. For the simplest

Table 1. *Evaluation of techniques for the structure test.*

Technique	Time (sec.)	Threshold	Classified as copies	Actual copies	Recall	Precision
τ_1	12	1.00	5	5	0.19	1.00
		0.95	7	7	0.27	1.00
		0.90	11	9	0.35	0.82
		0.85	11	9	0.35	0.82
τ_2	49	1.00	5	5	0.19	1.00
		0.95	7	7	0.27	1.00
		0.90	11	9	0.35	0.82
		0.85	11	9	0.35	0.82
τ_3	94	1.00	5	5	0.19	1.00
		0.95	7	7	0.27	1.00
		0.90	15	11	0.42	0.73
		0.85	16	11	0.42	0.69
τ_4	169	1.00	53	13	0.50	0.25
		0.95	113	21	0.81	0.19
		0.90	174	25	0.96	0.14
		0.85	266	25	0.96	0.09

technique τ_1, precision is maximal and recall is minimal. When going from the simpler techniques to the more complex ones, precision decreases while recall increases. This observation nicely illustrates how precision and recall can be controlled by `Kato`'s system of composable camouflage techniques. Furthermore, the results indicate also that each of our considered camouflage techniques is used by students, although some of them occur usually in combination with others. For example, students who renamed auxiliary predicates usually also rename variables. According to our evaluation, many students who change the names of auxiliary predicates also perform other changes, such as reordering commutative operators or renaming variables.

Concerning the different plagiarism detection strategies realised in `Kato`, our experience suggests that the structure test provides the most accurate results when students spent some effort to camouflage the copy. The LCS program test, on the other hand, revealed exact or almost exact copies, while the LCS comment test seems to be a good additional indicator for plagiarism. The fingerprint test provided a fast way to detect exact copies because of identical hash values. However, it is to mention that the exercises were rather rigorously specified which implies that it was rather common that programs agreed on some attributes like, e.g., the number of different predicate symbols.

7 Conclusion

In this paper, we presented the tool `Kato` for plagiarism detection in ASP, reviewing its underlying methodology and its basic features. In particular, we introduced the formal basis of `Kato`: syntactic notions of program similarity between programs along with a formal measure of confidence regarding plagiarism detection for programs. Both concepts are based on the notion of a camouflage technique which is basically a strategy to disguise the form of a program and which provides a flexible means to deal with different strategies and combinations thereof in a uniform manner. Currently, `Kato` is designed for `DLV`'s language dialect but it can easily be extended to other dialects. For `DLV` programs, `Kato` supports also weak constraints and aggregates, and it can also deal with programs for the planning front-end of DLV (Eiter *et al.* 2003). The system was successfully applied for laboratory courses at our university and helped to save valuable labour time.

For future work, we plan to develop means to visualise the comparison results, e.g., to spot clusters of cooperating plagiarists more easily. A further aspect of `Kato` worth mentioning is a possible use for the development of logic programs: If a team is working on a program, different versions can emerge. Then, assessing the actual differences between two versions can be necessary. `Kato` can be adapted for such an application scenario.

References

Arwin, C. and Tahaghoghi, S. M. M. 2006. Plagiarism detection across programming languages. In *Proceedings of the Twenty-Ninth Australasian Computer Science Conference* (*ACSC 2006*). CRPIT, vol. 48. Australian Computer Society, Inc., Darlinghurst, Australia, 277–286.

Austin, M. and Brown, L. 1999. Internet plagiarism: Developing strategies to curb student academic dishonesty. *The Internet and Higher Education 2,* 1, 21–33.

Bergroth, L., Hakonen, H. and Raita, T. 2000. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval* (*SPIRE 2000*). IEEE Computer Society, Los Alamitos, California, 39–48.

Clough, P. 2000. 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.

Denecker, M., Vennekens, J., Bond, S., Gebser, M. and Truszczynski, M. 2009. The second answer set programming competition. In *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2009*). Lecture Notes in Computer Science, vol. 5753. Springer, Berlin, 637–654.

Eiter, T., Faber, W., Leone, N., Pfeifer, G. and Polleres, A. 2003. A logic-programming approach to knowledge-state planning II: The DLV^{K} system. *Artificial Intelligence 144,* 2, 157–211.

Farringdon, J. M. 1996. *Analyzing for Authorship: A Guide to the Cusum Technique*. University of Wales Press, Cardiff.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3/4, 365–385.

Gitchell, D. and Tran, N. 1999. Sim: A utility for detecting similarity in computer programs. In *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 1999*). SIGCSE Bulletin, vol. 31. ACM Press, New York, 266–270.

Halstead, M. H. 1977. *Elements of Software Science*. Elsevier Science Inc., New York.

Jones, E. L. 2001. Metrics based plagiarism monitoring. *Journal of Computing Sciences in Colleges 16,* 4, 253–261.

Lancaster, T. and Culwin, F. 2005. Classifications of plagiarism detection engines. *ITALICS 4,* 2, `url:http://www.ics.heacademy.ac.uk/italics/vol4iss2.htm`.

Loose, F., Becker, S., Potthast, M. and Stein, B. 2008. Retrieval-Technologien für die Plagiaterkennung in Programmen. Tech. rep., University of Würzburg, Germany.

Lukácsy, G. and Szeredi, P. 2005. A generic framework for plagiarism detection in programs. In *Proceedings of the 4th Japanese-Hungarian Symposium on Discrete Mathematics and Its Applications*. 189–198.

Lukácsy, G. and Szeredi, P. 2009. Plagiarism detection in source programs using structural similarities. *Acta Cybernetica 19,* 1, 191–216.

MAURER, H., KAPPE, F. AND ZAKA, B. 2006. Plagiarism: A survey. *Journal of Universal Computer Science 12,* 8, 1050–1084.

MOZGOVOY, M. 2008. *Enhancing Computer-Aided Plagiarism Detection*. VDM Verlag, Saarbrücken, Germany.

OTTENSTEIN, K. J. 1976. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bulletin 8,* 4, 30–41.

PRECHELT, L., MALPOHL, G. AND PHILIPPSEN, M. 2000. *JPlag: Finding Plagiarisms Among a Set of Programs*. Technical Report 2000-1, Fakultät für Informatik Universität Karlsruhe, Germany.

SEREBRENIK, A. AND VANHOOF, W. 2007. Fingerprinting logic programs. *CoRR abs/cs/0701081*.

VERCO, K. L. AND WISE, M. J. 1996a. Plagiarism a la mode: A comparison of automated systems for detecting suspected plagiarism. *The Computer Journal 39,* 9, 741–750.

VERCO, K. L. AND WISE, M. J. 1996b. YAP3: Improved detection of similarities in computer program and other texts. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 1996*). ACM Press, New York, NY, USA, 130–134.

WHALE, G. 1990. Identification of program similarity in large populations. *The Computer Journal 33,* 2, 140–146.

WISE, M. J. 1993. *Running Karp-Rabin Matching and Greedy String Tiling*. Technical report, Basser Department of Computer Science, University of Sydney, Australia.