Token-based Plagiarism Detection for Metamodels

Timur Sağlam, Sebastian Hahner, Jan Willem Wittler, Thomas Kühn
{timur.saglam,sebastian.hahner,jan.wittler,thomas.kuehn}@kit.edu
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

ABSTRACT

Plagiarism is a widespread problem in computer science education. Manual inspection is impractical for large courses, and the risk of detection is thus low. Many plagiarism detectors are available for programming assignments. However, very few approaches are available for modeling assignments. To remedy this, we introduce token-based plagiarism detection for metamodels. To this end, we extend the widely-used software plagiarism detector JPlag. We evaluate our approach with real-world modeling assignments and generated plagiarisms based on obfuscation attack classes. The results show that our approach outperforms the state-of-the-art.

CCS CONCEPTS

• **Software and its engineering** → *Model-driven software engineering*; • **Social and professional topics** → *Software engineering education*; • **Information systems** → **Near-duplicate and plagiarism detection**;

KEYWORDS

Plagiarism Detection, Token-based Plagiarism Detection, Metamodeling, Metamodel Similarity, Obfuscation Attacks, Education, JPlag

1 INTRODUCTION

Plagiarism is widespread in computer science education [3, 6, 11]. Despite the threat of penalties, some students still try to cheat. This problem especially persists in computer science, as digital submissions can be copied with minimal effort. Students creatively obfuscate their plagiarism, e.g., by renaming, reordering, or restructuring [9, 12]. This is prevalent for mandatory assignments, e.g., in beginners' courses. As computer science classes are often large, manual inspection is impractical [4, 7]. Large classes lower the individual risk of detection [23]. To this end, automated plagiarism detection can be employed [13, 14]. A plagiarism detector aims to identify pairs of similar sections in two programs. From all matching sections, the detector calculates a similarity score. Confirming candidates as actual plagiarism is then always up to the instructors. Most plagiarism detectors, however, are optimized for plagiarism in code submissions. The state-of-the-art approaches are token-based and extract and compare the structure of the code [12]. Token-based plagiarism detection combines tokenization, normalization, and a comparison algorithm based on hashing (e.g., winnowing [15]) or string tiling (e.g., GST-RKRM [22]). Plagiarisms are found by extracting tokens from the programs' syntax tree, where the similarity of two programs is determined by matching their token sequences. Only a subset of the tree nodes is extracted as tokens, which acts as an abstraction from the code. Therefore, detectors like JPlag [14], MOSS [1, 15], or Sherlock [8] are resilient against typical obfuscation techniques, like renaming or reordering [14].

However, assignments in computer science often also include modeling tasks [5], e.g., creating a suitable metamodel for a given domain. While model comparison and clone detection approaches exist, they are not directly applicable to metamodel plagiarism detection. *Clone detection* [16] deals with finding identical code fragments in a program. While it is closely related to plagiarism detection, they are different problems as plagiarism detectors must face the threat of obfuscation attacks. For clone detection, the size of the modification to the clone should be reflected in the similarity measures. In contrast, for plagiarism detection, it is even valid that some changes will not reduce the similarity at all to achieve resilience against obfuscation attacks. Furthermore, false positives are exceptionally problematic for plagiarism detection and should be avoided at all costs. While we thus relate to metamodel clone detection [17, 20], like the approach by Babur et al. [2], these approaches are not sufficient for plagiarism detection as they are prone to typical obfuscation attacks. Additionally, we relate to *Model Differencing* [19], as it also calculates the similarity between models. However, it is again susceptible to obfuscation attacks.

This paper presents an approach for applying token-based plagiarism detection to metamodels. We extend the state-of-the-art [21] plagiarism detector JPlag [14]. Thus, we inherit benefits like scalability, usability, or resilience against common obfuscation attacks and a graphical interface. We employ a real-world modeling assignment as our case study to evaluate our approach. Next, we generate plagiarisms by applying multiple classes of obfuscation attacks [9, 12], like element insertion or element swapping. We compare the accuracy of our plagiarism detector with the state-of-the-art approach by Martínez et al. [10], which is based on Locality Sensitive Hashing (LSH). The results show significantly better plagiarism detection capabilities than the LSH-based approach in most scenarios due to a higher resilience to obfuscation attacks. Our approach highly reduces the effort of manual inspection, enabling plagiarism detection of modeling tasks even for large classes.

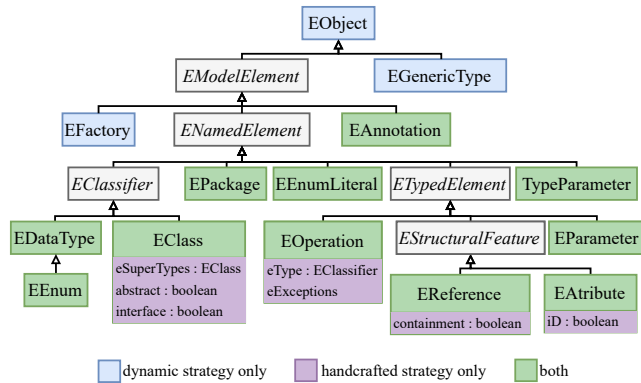dynamic strategy only · handcrafted strategy only · both

Figure 1: The type hierarchy of the Ecore meta-metamodel, showing which information is extracted by the two strategies.

2 TOKENIZATION OF METAMODELS

Similar to code, metamodels can describe a program's structure and behavior. This similarity between metamodels and code can be exploited to enable token-based plagiarism for modeling artifacts. To this end, we extend the software plagiarism detector JPlag [14]. We chose JPlag, as it is open-source[1], can be easily extended, offers good scalability, and is widely used. JPlag supports multiple programming languages by providing different language modules. Each language module needs to fulfill two responsibilities: First, it parses the code and transfers it into a tree structure, e.g., a parse tree or an abstract syntax tree. Second, the language model extracts a sequence of tokens from the nodes of the tree structure. Each token only contains the structural information of its corresponding node type. For example, a variable declaration token does not preserve the identifying name or the declared type.

To leverage JPlag for metamodels, we design a language module for metamodels. This language model extracts a token sequence based on the corresponding meta-metamodel. This extraction step requires less abstraction than a language module for code because metamodels are already implementation-independent conceptual models. Consequently, the first responsibility of parsing the metamodel into a tree structure is trivial, as metamodels already provide a tree structure. The metamodel tree can be traversed from its root via its containment references. The second responsibility is extracting the token sequence from the metamodel elements. The tokens need to describe the essence of the metamodel but need to filter out details like element names and types. Thus, we define a token set and corresponding extraction rules based on the metaclasses of the meta-metamodel. The token set and the extraction rules specify which and how model elements are extracted as tokens. The extracted token sequence is then passed to the JPlag comparison algorithm. We present two token selection strategies: First, a naive strategy based on a dynamically created token set. Second, a specialized strategy based on a handcrafted token set. Both strategies were implemented for EMF [18] metamodels and thus based on the Ecore metaclasses. However, both strategies can be easily implemented for different modeling frameworks.

---
[1]https://www.github.com/jplag

2.1 Dynamic Token Selection

For the *dynamic token selection*, we define the set of possible tokens based on the concrete metaclasses of the meta-metamodel. To extract the tokens, we traverse the metamodel and extract tokens containing only the metaclass of the element as information. Properties like the element name are not transferred. This approach thus extracts tokens for packages, classifiers, attributes, references, etcetera. Figure 1 visualizes the inheritance tree of the Ecore meta-metamodel. The information extracted by the dynamic token selection is shown in blue and green. As the Ecore meta-metamodel is self-describing and self-instantiating, the dynamic token selection can be applied, in addition to metamodels, to their instances. Since we focus on metamodels, we did not investigate this in detail. However, the dynamic token selection comes with two downsides. As a first problem, it also extracts tokens with very little meaning. We observed EGenericType to be especially problematic, as EMF automatically creates an EGenericType for every ETypedElement in a metamodel. As a second problem, some vital information is not transferred into tokens. EMF distinguishes concrete classes, abstract classes, and interfaces via two flags stored as attributes in the metaclass EClass. Thus, the dynamic token selection only extracts class tokens and cannot make a more precise distinction. Analogously, containment references are not distinguished from regular references, and identifier attributes are not distinguished from regular attributes.

2.2 Handcrafted Token Selection

The *handcrafted token selection* builds upon the previous strategy. It also uses metaclasses to extract tokens. However, it employs a fixed set of tokens. Compared to the dynamic strategy, there are three key differences in the token set: *1. Stricter Token Selection.* The concrete metaclasses EFactory, EGenericType, and EObject are not extracted as tokens. They provide superfluous information as they are rarely explicitly modeled by a student. Thus, they needlessly increase the attack surface by allowing changes to the token sequence that do not alter the semantics of the metamodel. *2. Token Distinction via Attributes.* Some concrete metaclasses are further distinguished to extract more fine-grained information. An EClass can be a class, abstract class, or interface. Containment references have a different token type compared to non-containment ones. Analogously, identifier attributes differ from normal attributes. This distinction broadens the token set and thus reduces false positives, as the discerned model elements semantically serve very different purposes. *3. Extraction from Meta-References.* Some tokens are extracted for important meta-references in the Ecore meta-metamodel. Each superclass reference of an EClass is extracted as *super type* token, a return type reference of each non-void EOperation is extracted as *return type* token. Again, these additional tokens reduce false positives. Moreover, they allow detecting additional similarities, like the number of declared super types. As depicted in Figure 1, fewer concrete metaclasses are used for tokens. However, additional information is used to distinguish more token types. In contrast to the dynamic token selection, this handcrafted strategy is tailored explicitly towards metamodels and thus not applicable to their instances. However, while it tends to extract fewer tokens than the dynamic strategy for the same input, it can differentiate between more token types through its refined and extended token set.

Table 1: Obfuscation attacks to plagiarize metamodels. Each operation is executed ten times for each marked type.

Operation	Package	Class	Operation	Attribute	Feature	Reference	Supertype	Level [9]	Type [2]
Insert	✓	✓	✓	✓		✓	✓	L2.5, L3, L4	B, C
Delete		✓		✓		✓	✓	-	B
Move	✓	✓			✓			L2.5, L3	B, C
Swap	✓	✓			✓			L2.5, L3	B, C
Rename	✓	✓		✓		✓		L2, L2.5	C

2.3 Minimum Match Length

JPlag allows adjusting the detection process via multiple parameters. Most notably, the *minimum match length* (MML) specifies the minimum length for two code sections to be treated as a match. Thus, the MML controls the sensitivity of the comparison. Lowering the MML increases the sensitivity for plagiarism but also increases the number of false positives. Based on our initial experiments, we chose the default MML values 10 for the dynamic strategy and 6 for the handcrafted one. The dynamic strategy has a higher default value, as it generally produces around 1.5 times as many tokens for the same model. For different datasets based on different modeling tasks, the ideal MML might vary and can thus be adjusted.

3 CASE-STUDY BASED EVALUATION

We evaluate our approach based on 18 metamodels from real-world assignments. We systematically generated 80 plagiarisms based on the metamodels by applying obfuscation attacks. With this data, we compare our approach with both token selection strategies against the LSH-based plagiarism detector of Martínez et al. [10].

3.1 Evaluation Design

Our data set contains metamodels from modeling assignments from a master's level elective practical course on model-driven software development. The assignment tasks the students with creating a metamodel for designing component-based system architectures, which allows the creation of models like UML component diagrams but also involves the aspect of software-to-hardware allocation. Students solved the same modeling assignment in small groups, 18 of whom consented to us using their metamodels. These 18 metamodels are, to the best of our knowledge, free of plagiarism. On average, the metamodels contain five packages, 39 classifiers, 45 references, ten attributes, and one operation. We generate plagiarized metamodels by copying original metamodels and applying obfuscation attacks from existing classifications [2, 9]. We randomly chose four original metamodels and applied the same 20 attacks for each to create a total of 80 plagiarized metamodels.

Our attack set is based on existing works regarding obfuscation attack [2, 9, 20]: We omit trivial attacks (Type-A [2], L0/L1 [9]) like verbatim copying, as these do not change the token sequence. Additionally, we omit attacks against which our approach and the LSH-based approach are inherently resilient. This includes attacks like changing an attribute's type or a reference's multiplicity and trivial name changes, e.g., typos. In line with [2], we also omit some more complex attacks, namely Type-D semantic clones. Table 1 shows the complete attack set we use for our evaluation. We executed these modifications on different element types, e.g., classes,

attributes, and references. For each attack, we executed the modification on ten random model elements of the same type. For the insertion of references and supertypes, we referenced random existing classes. For renames, we generated realistic names based on the existing names in the metamodel, which are indistinguishable at first glance. As the metamodels contain few to no operations, we only conducted the insertion of operations since the other attacks would have had little to no effect on the metamodels. We also did not conduct deletion of packages, as this completely breaks the metamodels and is thus no viable attack. For JPlag, we use the respective default MML for each strategy. For fairness, we did not conduct any parameter tuning with the evaluation data set. For the LSH-based approach [10] we used the parameters they provided in their implementation. In particular, $b = 5$ and $r = 30$.

3.2 Evaluation Results and Discussion

Figure 2 shows the results for the different attack groups from Table 1. We compare JPlag with the dynamic token selection (*JPlag-D*) and the handcrafted token selection (*JPlag-H*), as well as the LSH-based approach [10] (*LSH*). We show the similarity distribution for plagiarism and unrelated original tuples for each approach. The similarity should be high for the plagiarism tuples and low for the original tuples. The larger this difference, the higher the accuracy of the plagiarism detection. If these tuple types overlap, identifying all plagiarism tuples is not possible.

For the insertion attacks (2a), JPlag performs well. LSH performs well, except for attribute and operation insertions. For these attacks, the accuracy is low enough that identifying plagiarism becomes less exact. For the deletion attacks (2b), both JPlag and LSH struggle with class deletion attacks. However, JPlag-H shows the highest accuracy. Both perform well for the deletion of references and the removal of supertypes. However, for the deletion of attributes, LSH shows low accuracy and a slight overlap between both tuple groups. Both JPlag and LSH perform very well for moving and swapping attacks (2c). JPlag is more sensitive against swapping and moving classes than LSH but still achieves sufficiently high accuracy. For the renaming attacks (2d), renaming packages and references does neither affect JPlag nor LSH. However, when renaming classifiers and attributes, LSH has low accuracy. Especially for renamed classifiers, plagiarisms cannot clearly be distinguished from non-plagiarisms. In total, LSH shows a significant drop in accuracy for particular attacks. The reduced accuracy is noteworthy as each attack only executes ten modifications, which is easy to do for a student. JPlag performs very well for renaming, as it is resilient against such attacks by design. While LSH performs better than JPlag for some attacks, JPlag still achieves reasonably high accuracy in these cases. For attacks against packages, LSH always produces 100% similarity, as it only compares classes and their contents and is thus inherently unable to detect changes to other elements. Both selection strategies for JPlag have high accuracy, but JPlag-D determines a higher similarity for the original tuples, making the plagiarism identification less exact.

We now discuss threats to validity and limitations. Regarding *internal validity*, real plagiarized metamodels were unavailable, so we created the plagiarized metamodels ourselves. To counteract subconscious bias, we designed a fully-automated generation process that randomly chooses metamodels and applies obfuscation

(a) Element Insertion

(b) Element Deletion/Removal

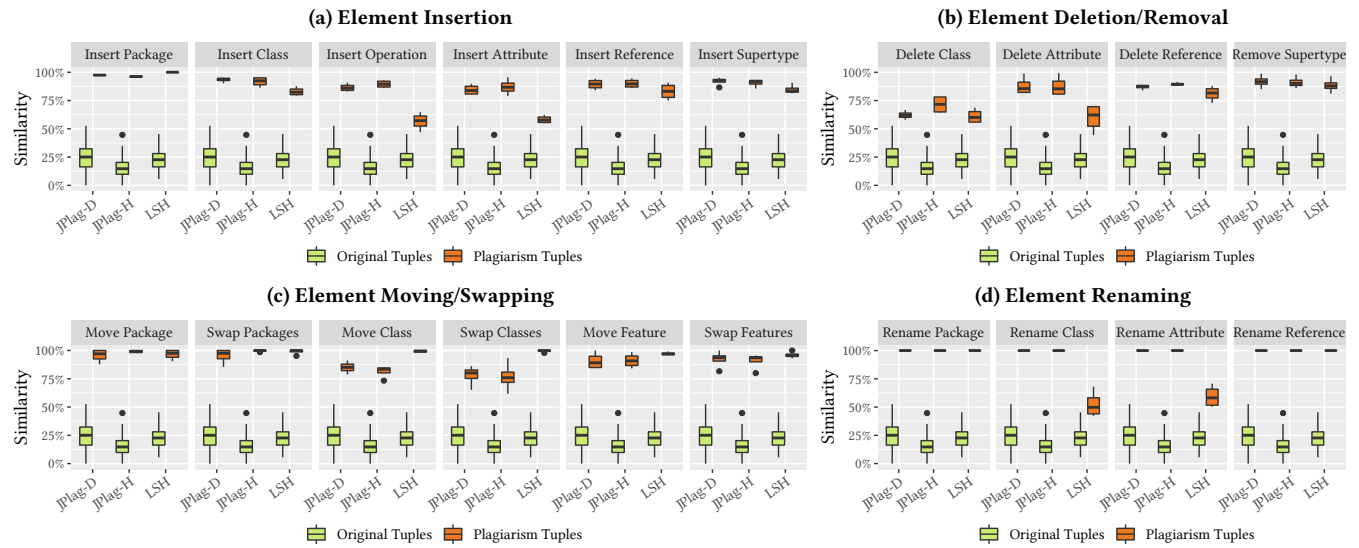**(c) Element Moving/Swapping**

(d) Element Renaming

Figure 2: Evaluation results for JPlag with the dynamic token selection (JPlag-D), JPlag with the handcrafted token selection (JPlag-H), and the LSH-based approach by Martínez et al. [10] (LSH).

attacks from existing work [2, 9]. For *external validity*, we only apply one attack type at a time to investigate the effects of each type in isolation. However, simple obfuscation attacks are typical for students [12]. To maximize *construct validity*, we performed obfuscation attacks frequently encountered in reality [9, 12] on metamodels from real-world modeling assignments. However, the *reliability* is limited due to the sensitive nature of our data set, which cannot be published. Nevertheless, our methodology's description allows re-applying it to other datasets. Regarding *limitations*, our approach produces less conclusive results for very small models. However, this is an inherent problem of plagiarism detectors as the space of possible solutions collapses with decreasing model sizes. Our approach currently only supports metamodels, but the dynamic token selection could also be applied to their instances.

4 CONCLUSION

In this paper, we introduce token-based plagiarism detection for metamodels. We extend JPlag [14] to enable the tokenization and plagiarism detection of EMF metamodels. Our approach is resilient to typical obfuscation attacks such as renaming and retyping. We evaluate our approach with real-world assignments and plagiarism based on obfuscation attacks from literature [2, 9]. The results not only show the feasibility of our approach but also that it performs significantly better than the state-of-the-art [10]. In the future, we want to extend our approach for instances of metamodels, and evaluate it with other data sets, e.g., by Martínez et al. [10].

ACKNOWLEDGMENTS

REFERENCES

[1] Alex Aiken. 2022. *MOSS Software Plagiarism Detector Website.* Stanford University. http://theory.stanford.edu/~aiken/moss/ Accesed: 2022-07-20.
[2] Önder Babur et al. 2019. Metamodel clone detection with SAMOS. *COLA* 51 (2019), 57–74.
[3] Jess Bidgood and Jeremy B. Merrill. 2017. As computer coding classes swell, so does cheating. *The New York Times* (2017). https://www.nytimes.com/2017/05/29/us/computer-science-cheating.html Accessed: 2022-07-21.
[4] Tracy Camp et al. 2017. Generation CS: The Growth of Computer Science. *ACM Inroads* 8, 2 (2017), 44–50.
[5] Federico Ciccozzi et al. 2018. How Do We Teach Modelling and Model-Driven Engineering? A Survey. In *MODELS-C*. ACM, 122–129.
[6] Georgina Cosma and Mike Joy. 2008. Towards a Definition of Source-Code Plagiarism. *IEEE Transactions on Education* 51, 2 (2008), 195–200.
[7] Breanna Devore-McDonald and Emery D. Berger. 2020. Mossad: Defeating Software Plagiarism Detection. *PACMPL* 4 (2020), 1–28.
[8] Mike Joy and Micheal Luck. 1999. Plagiarism in programming assignments. *IEEE Transactions on Education* 42, 2 (1999), 129–133.
[9] Oscar Karnalim. 2016. Detecting source code plagiarism on introductory programming course assignments using a bytecode approach. In *ICTS*. IEEE, 63–68.
[10] Salvador Martínez et al. 2020. Efficient plagiarism detection for software modeling assignments. *Computer Science Education* 30, 2 (jan 2020), 187–215.
[11] William Murray. 2010. Cheating in Computer Science. *Ubiquity* 2010 (2010).
[12] Matija Novak et al. 2019. Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *TOCE* 19, 3 (2019), 1–37.
[13] K. J. Ottenstein. 1976. An Algorithmic Approach to the Detection and Prevention of Plagiarism. *SIGCSE Bulletin* 8, 4 (1976), 30–41.
[14] Lutz Prechelt et al. 2002. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science* 8, 11 (2002).
[15] Saul Schleimer et al. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *SIGMOD*. ACM, 76–85.
[16] G. Shobha et al. 2021. Code Clone Detection—A Systematic Review. In *IEMIS*. Springer Nature Singapore, 645–655.
[17] G Shobha et al. 2021. Comparison between Code Clone Detection and Model Clone Detection. In *ICRITO*. IEEE, 1–5.
[18] David Steinberg et al. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.
[19] Matthew Stephan and James R. Cordy. 2013. A Survey of Model Comparison Approaches and Applications. In *MODELSWARD*. INSTICC, SciTePress, 265–277.
[20] Harald Störrle. 2015. *Effective and Efficient Model Clone Detection.* Springer, 440–457.
[21] Debora Weber-Wulff et al. 2012. Collusion detection system test report 2012. *Hochschule für Technik und Wirtschaft, Berlin, Tech. Rep* (2012). Technical Report.
[22] Michael Wise. 1993. String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. *Basser Departement of Computer Science Report* (01 1993).
[23] Lisa Yan et al. 2018. TMOSS: Using Intermediate Assignment Work to Understand Excessive Collaboration in Large Classes. In *SIGCSE*. ACM, 110–115.