Efficient Plagiarism Detection for Software Modeling Assignments

Salvador Martínez[a], Manuel Wimmer[b] and Jordi Cabot[c]

[a]IMT Atlantique - Lab-STICC, Brest, France;
[b]CDL-MINT, Johannes Kepler University, Linz, Austria;
[c]ICREA-UOC, Barcelona, Spain

ARTICLE HISTORY
Compiled January 8, 2020

ABSTRACT
Background and Context: Reports suggest plagiarism is a common occurrence in universities. While plagiarism detection mechanisms exist for textual artifacts, this is less so for non-code related ones such as software design artifacts like models, metamodels or model transformations.
Objective: To provide an efficient mechanism for the detection of plagiarism in repositories of Model-Driven Engineering (MDE) assignments.
Method: Our approach is based on the adaptation of the Locality Sensitive Hashing, an approximate nearest neighbor search mechanism, to the modeling technical space. We evaluate our approach on a real use case consisting of two repositories containing 10 years of student answers to MDE course assignments.
Findings: We have found that: (*i*) effectively, plagiarism occurred on the aforementioned course assignments (*ii*) our tool was able to efficiently detect them.
Implications: Plagiarism detection must be integrated into the toolset and activities of MDE instructors in order to correctly evaluate students.

KEYWORDS
Model-Driven Engineering; Robust Hashing; Locality Sensitive Hashing; Clustering; Plagiarism Detection;

1. Introduction

Model-driven engineering (MDE) is a software engineering approach that considers models as first-class citizens of the development process (Brambilla, Cabot, & Wimmer, 2017). As such, models can be used in all phases of the development process and in a variety of scenarios including, for instance, early verification and testing or even (semi)automatic code generation. The increased adoption of this paradigm (see Whittle, Hutchinson, & Rouncefield, 2014) in both, academic and industrial scenarios comes together with a proliferation of new MDE courses and programs (Ciccozzi et al., 2018) in computer science and engineering schools in order to respond to the needs and demands of students and industry. Many of these courses are an evolution of the more "traditional" systems and software analysis and design courses. In those courses, plagiarism of modeling assignments was also an issue but this challenge is more exacerbated in

CONTACT Salvador Martínez. Email:salvador.martinez@imt-atlantique.fr

MDE courses where models are not just one more element in the development process but the driving force of such process.

As reported by Ciccozzi et al. (2018), most of the existing MDE courses include assignments where students are required to build modeling artefacts. In many cases (up to 76% of the courses), these assignments are the only means to evaluate the students. Therefore a fair assessment of these artefacts is key for a proper student evaluation.

However, reports suggest plagiarism is a common occurrence in universities (Blum, 2009; Starovoytova & Namango, 2016). Students usually recur to public repositories or copies of previous year assignments in order to complete all or part of their tasks. This is also the case for computer education assignments as evidenced by the plagiarism detection efforts discussed in Joy and Luck (1999), Lancaster and Culwin (2004) and Ďuračík, Kršák, and Hrkút (2017).

Even if modeling and, especially, full-fledged MDE, is a somehow recent field, there are already a number of large courses, repositories and solved exercises online that could be used for plagiarism. Indeed, in some cases, the number of editions of the same course may be high (see Ciccozzi et al., 2018). Additionally, and although we focus on a-posteriori measures (i.e., detection of already occurred plagiarism), Joy and Luck (1999) and Rosales et al. (2008) show that plagiarism detecting tools act as a deterrent making the students less prone to incur in plagiarism, and thus, may improve academic integrity. Thus, in order to effectively asses the outcomes of MDE programs, plagiarism detection must be integrated into the toolset and activities of instructors.

Unfortunately, while ready-to-use plagiarism detection mechanism exist for textual documents, there is nothing really useful for plagiarism detection on MDE artefacts. Approaches proposed for classical computer programming language assignments (see Bejarano, García, & Zurek, 2015; Inoue & Wada, 2012; Lancaster & Culwin, 2004; Rosales et al., 2008) while specially designed to deal with academic assignments (e.g., they can deal with artefacts that are smaller than their real life counterpart and often more similar among them), are not directly usable at the modeling level, where we need specific comparison mechanisms. The few MDE-specific approaches available typically require pairwise comparison of models, too computationally expensive.

To address these issues, we provide in this paper an efficient mechanism for the detection of plagiarism in repositories of MDE assignments based on an adaptation of the Locality Sensitive Hashing (LSH) (Indyk & Motwani, 1998) technique to the modeling technical space. LSH is an approximate nearest neighbour search mechanism successfully used for clustering. Effectively, using LSH over a (model) repository results in its classification (without the use of pairwise comparisons) in a set of buckets which basically constitute a set of similarity based (modeling) clusters.

We demonstrate the feasibility of our approach by providing a prototype tool implementation (available online). We evaluate it on a real use case consisting of two repositories containing 10 years of student answers to MDE course assignments corresponding to lessons related to modeling (and metamodeling) and model transformation taught between 2007 and 2017. As a result, we have found that: (*i*) effectively, plagiarism occurred on the aforementioned course assignments (*ii*) our tool was able to efficiently detect them, and thus, may be used by instructors to asses the outcomes of their courses.

The rest of the paper is organized as follows. Section 2 presents background on the main elements of our approach, namely, model-driven engineering concepts and its academic training, robust hashing and LSH. Section 3 describes how do we use robust hashing and LSH in order to build our plagiarism detection tool. Evaluation's use case and results are presented in Section 4. Related work is discussed in Section 5. We finish

the paper in Section 6 by providing conclusions and an outlook on future work.

2. Background

This section presents some preliminary concepts on Robust Hashing and LSH needed to understand how both techniques are extended and adapted to built our plagiarism detection mechanism for MDE. Basic MDE terminology as well as a short overview on teaching MDE courses is also provided at the end of this section.

2.1. Robust Hashing

Classical cryptographic hashes such as SHA1 or MD5 may be used for the authentication and integrity assessment of digital assets. However, and due to the avalanche effect they include in their design, small changes to the asset lead to the generation of very different hashes, making them unsuitable for other tasks such as fast comparison and retrieval, intellectual property protection or plagiarism detection. This is so because in these scenarios we are interested not only in finding exact assets, but also variations of the assets.

In order to solve this problem, the concept of robust (or perceptual) hashing has been introduced, notably in the domain of digital images (Fridrich & Goljan, 2000) but also in other domains such as those of 3D mesh models (Lee & Kwon, 2012) and textual documents (Steinebach, Klöckner, Reimers, Wienand, & Wolf, 2013). A robust hash is a hash that can resist a certain type and/or number of data manipulations. This is, the hash obtained from a digital asset and that from another asset similar to the original one but that has been subjected to minor manipulations should be the same or at least very similar. As an example, robust hash algorithms for images or 3D model mesh resist manipulations such as rotation and compression, as they remain visually *similar*. Text documents remain similar if they convey the same message, thus, they resist to attacks introducing synonyms, etc.

Summarizing, robust hash algorithms transform information from a certain data type to hashes that preserve the proximity between data type instances under a chosen distance measure.

2.2. LSH algorithm for nearest neighbour search

The aforementioned robust hashing technique transforms complex data into smaller, easier to manage hashes, which already helps in similarity based search and classification tasks. However, when datasets are large the bottleneck is not only in the complexity of the comparison of two data instances, but in the increasing number of pairwise comparisons required to perform any search or classification task. This latter problem may be solved by the use of locality sensitive hashing techniques.

Locality-sensitive hashing (LSH) was first introduced by Indyk and Motwani (1998) as a method for finding approximate nearest neighbours for highly dimensional data. The method first selects a family of hash functions for which the probability of a collision is high if the hashed objects are similar (given a distance measure, d and a threshold R). If objects are dissimilar, the hash functions are very likely to hash them to separate buckets. Now, to find near-neighbours of a query point, one hashes that point with each of the hash functions and returns the elements stored in the buckets the point gets

hashed to.

Informally, the requirements to apply the LSH technique to a given type of data are as follows:

- The type of data to which we intend to apply LSH must form a *metric space*. Basically, this means that there must be a metric function that defines a concept of distance between any two members of the set containing the elements of the data type (e.g., if our data type is sets, we need a metric function that allow us to calculate the *distance* between any two sets).
- A family of locality-preserving hash functions which map elements from the metric space to a bucket $s \in S$ must exist. Particularly we are interested in (d_1, d_2, p_1, p_2)-sensitive families. With $d_1 < d_2$ being two distances in a given metric space, a family F is (d_1, d_2, p_1, p_2)-sensitive if: (i) the probability of collision when the distance between two elements is less that d_1 is at least p_1; (ii) the probability of a collision when the distance is higher than d_2 is smaller that p_2.

Summarizing, a LSH scheme for a given data type allows us to find (with high probability) similar elements in a given repository using computationally efficient hashing functions instead of pairwise comparisons.

2.3. Model Driven Engineering

Model-driven engineering (MDE) is a software engineering approach that considers models as first-class citizens of the development process (Brambilla et al., 2017). In an abstract way, models can be regarded as structured data (the structure is defined by the metamodel the model conforms to) composed of model elements that contain a set of attributes and/or reference slots (other models may be then built by using this basic building blocks). Similarly, metamodels conform to metametamodels and, as such, they can be regarded (and manipulated) as models as well; In fact, all MDE artefacts (e.g., model transformation and model queries) can be represented as models themselves and thus, can be uniformly treated (Bézivin, 2005).

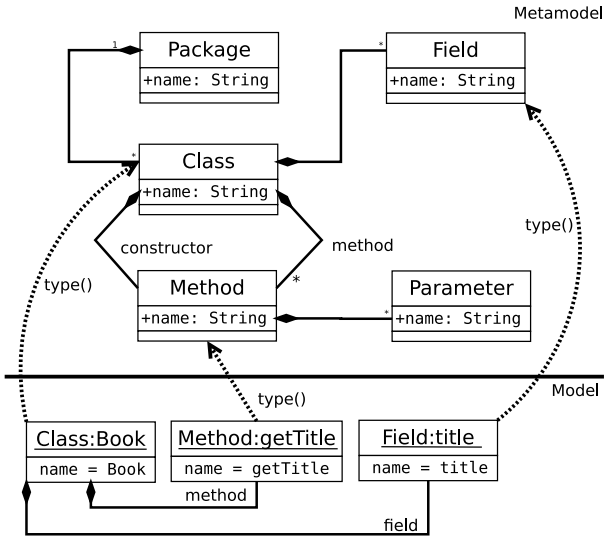


Figure 1. Model and Metamodel Example

As an example we show in Figure 1 a simplified Java metamodel (upper part) and a corresponding model (lower part). The Java metamodel establishes that Packages contain classes that in turn contain fields and methods (including an special method named constructor) with parameters. The Java model declares a class named *Book* with a title field and a *getTitle* operation. Note that we use the notation Type:id (for simplicity id is equal to the element name) to distinguish metamodel classes from model classes in the diagrams in this paper.

2.4. Teaching Model Driven Engineering

With the rise of MDE, more and more dedicated modeling courses have been established at different universities worldwide, e.g., cf. the MDSE Book website[1] which lists over 100 MDE courses as well as Ciccozzi et al. (2018) for a recent survey on the state of the art of teaching MDE. In this mentioned survey, the main contents of 47 courses have been collected and reported. We shortly summarize the main results of this survey which are also of importance for this paper.

The majority of MDE courses are offered in master programs. Of course, this allows to build on already existing knowledge about software engineering and software modeling. However, there are also courses already offered in the bachelor programs. Some courses are offered for PhD students or for a mix of different types of students. Most courses are lectures combined with laboratory activities—thus, a mix of theory and practice. The surveyed courses have a quite extensive number of students, whereas over 20% of the courses have more than 90 students.

There are mainly two types of MDE courses currently offered. The first type is about courses teaching MDE for a particular domain by using already existing MDE tools, e.g., MDE is applied for the development of distributed systems. The second type of courses is focusing on teaching how to employ existing frameworks to develop MDE tools for particular domains. In these lectures, the different concepts, techniques, tools, and practical approaches from the field of MDE are examined. This includes in particular language engineering and transformation engineering (Brosch, Kappel, Seidl, & Wimmer, 2009). For the first part, meta-modeling for developing the abstract syntax and concrete syntax (textual and graphical) of modeling languages are taught. For the second part, model transformation and code generation are used as the main techniques to generate software applications from models.

In accompanying labs, the students are working on practical assignments chosen from the topics of the lectures. This often includes the creation of self-designed metamodels, model transformations, and code generation facilities. By this, students gain practical experience with state-of-the-art MDE tools. It has to be mentioned that in Ciccozzi et al. (2018) the Eclipse Modeling Framework (EMF) is reported as the most used framework for modeling language design. ATL is reported to be the most used model transformation language. Because both are so frequently used in teaching MDE, they are also the subject of investigation for this paper as well.

3. Approach

Figure 2 summarizes our approach. We start by calculating the robust hashes of all the models in a given repository. Subsequently, we use LSH to *classify* the models in

[1] `www.mdse-book.com`

buckets. Finally, plagiarism candidates are taken from the collisions in the buckets and filtered by means of automatic assessments. The (few) remaining candidates are then manually evaluated to confirm the plagiarism. Clearly, this is much more efficient than having to cross-examine all models for plagiarism. We devote the rest of this section to provide a detailed description and rationale for the aforementioned steps.

3.1. Robust Hashing for Models

The first step of our approach (see an overview in Figure 2) is to transform complex models into an easier to manage representation that enables the use of LSH techniques. The key requirement of this new representation is to ensure that it preserves the similarity of the models. The more similar the original models are, the more similar this derived representation must be. Robust hashing functions, as described in Section 2, meet this requirement (among others such as being resistant to certain manipulations). Thus, we adopt here the robust hashing approach for models (Martínez, Gérard, & Cabot, 2018) were the authors adapted the *minhash* technique (Broder, 1997) in order to transform models into vectors of n symbols.

In short, obtaining a robust hash for a model is achieved in 3 steps:

Model Fragmentation. Model elements include individual data (attributes, operations) but are also related to a number of other model elements via their references. Both aspects need to be considered for a robust hashing. If we consider only the content, two models with the same elements would generate the same hash even if those elements were organized according to a very different structure. And, similarly, if we just take the structure into account, models of two very different domains that, by chance, share a similar structure, could be regarded as equivalent.Thus, as a first step, we create a number of (possibly overlapping) fragments from the model. This allows to generate the hashes using as unit not the single model element (which, as we have discussed before, would not be good enough) but the element together with some contextual information.

Model Fragment's Signatures Generation. Previously to the hashing of the extracted model fragments, we need to transform them to *summary sets*, this is, sets containing words representing the contents of the fragment. This enables us to subsequently use the *minhash* technique to obtain the fragment signatures. We use content-based identities of model elements (see Reddy, France, Ghosh, Fleurey, & Baudry, 2005) as the base for the translation from model fragments to summary sets. We call this content-based identities *model summaries* and *model fragment summaries*.

Classification of Model Fragments and Hash Sequencing. The two previous steps transform a model into a long set of minhash signatures. Indeed, a large number of signatures is created in order to avoid model mutations to influence the hash creation process (e.g., modifying the order of selected elements). For the generation of the final hash, a number of different signatures is chosen. We do so by first performing a classification step that groups together similar signatures. Then, we can just choose one of the elements of a given group as its representative. This process guarantees that: 1) we take different signatures to build the hash; and 2) mutations have a limited impact in the selection step.

The vectors produced by the robust hashing function explained above can be used alone as a means to estimate the similarity between models, and thus, can be used on behalf of the corresponding models in tasks related with similarity analysis. In the following, we see how they can be used as part of LSH-based classification process.

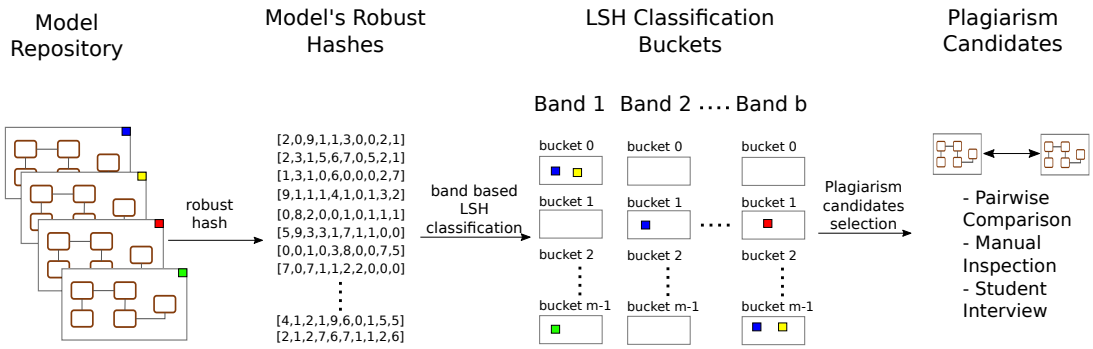


Figure 2. Our plagiarism detection approach, comprising a Robust Hashing and Band Classification Process

3.2. Locality Sensitive Hashing with Robust Model Hashes.

In classical partition clustering algorithms, pairwise comparisons between the feature vectors would be required. Instead, we intend to use Locality Sensitive Hashing to classify the model (represented by their vectors) without having to go through the very computationally expensive pairwise model comparison process.

As stated in Section 2, in order to apply LSH on a given type of data we need it to constitute a metric space (i.e., we need to define a distance metric that can be used to calculate the distance between two any elements of the same type) and to have a family of locality preserving hash functions for the same distance metric.

This is hard to achieve in models, as they are complex data structures and additionally: (*i*) no standard distance measure exists; (*ii*) no locality preserving function is defined for the domain. Arguably, we could use some of the techniques applied internally for the definition of the robust hash function for models described in the previous section in order to define a metric space and a family of hashing function for models. However, that would come with an important performance drawback, as hash functions must be defined and calculated on model fragments, which would require expensive model traversal and manipulation operations.

Thus, instead of the direct use of models, we propose here to deal only with their robust hashes. This way, we go from the modeling technical space to the vector space that forms a metric space under the *hamming distance*. An introduction to LSH for different metric spaces is given in Leskovec, Rajaraman, and Ullman (2014) from where we borrow notation and definitions. This transformation towards the vector metric space works because, as shown by Martínez et al. (2018), robust hashing vectors estimate well the similarity between models.

3.2.1. Band Based Classification

In general, it is easy to obtain a family of locality preserving hash functions for the vector metric space. As a example, any hash function taking one element of the vector would be such a hash function (and we can have as many as the size of the vectors) and we know from Leskovec et al. (2014) that they form a $(d_1, d_2, 1 - d_1/d, 1 - d_2/d) - sensitive$ family. These simple functions are not such good similarity estimators as they only take one point of the vector into account. However, the situation can be improved by building a new family of hash functions by using a technique called *amplification* (the amplification technique builds new families of hash functions by combining existing ones). The same effect can be obtained in a simpler way as described by Leskovec et

al. (2014): having vectors as data types, an effective way to perform the augmentation is to divide the vectors into b bands consisting of r rows each. For each band, there is a hash function (any standard hash function will work) that takes vectors of r integers and hashes them to some large number of buckets. Intuitively, this banding strategy makes similar robust hash vectors pairs much more likely to be hashed to the same bucket than dissimilar pairs. Basically, the banding process is mirroring the building of a $(d_1, d_2, 1-(1-p_1^r)^b, 1-(1-p_2^r)^b) - sensitive$ family. It can be seen that the probability of one band to have all rows equal to another is not very high. However, when this process is repeated for each of the bands, the probabilities get much higher As a example, with vectors of 200 elements and having two vectors with a similarity of 0.8, we pass from a probability of collision of 0.8 with the single point function to a probability of 0.989859579 when we use the banding strategy with $(1-((1-0.8^8)^{25}))$, and this without facilitating false positives.

In practice, the way we implement this band based classification is as follows:

(1) Dividing the robust hash vectors in bands (e.g., we could divide vectors of 200 elements into 8 bands of 25 elements each). Note that this process is critical. Bands should be sufficiently large so that false positives are unlikely but not so large that we are unable to get equal bands out of very similar vectors. This tuning depends on the typical similarity of the elements to be classified with the LSH technique.
(2) Rehashing the selected band by using any available hash function. This could be achieved with Java *hashcode* functions for vectors and modulo operations on the result. We need to have a different set of buckets for each band so that we avoid having inter-band collisions, that are meaningless.
(3) Tracking and aggregate collisions as the the bands are being hashed. The idea is to obtain at the end of the process a list of colliding robust hash pairs together with details on the colliding bands.

3.3. Copy Detection Validation

The main purpose of the LSH process is to determine if plagiarism occurred in student assignments. In order to validate that the collisions are indeed instances of plagiarism we need to perform the following steps: (*i*) *selection of candidates*; (*ii*) *automatic validation of candidates*; (*iii*) *manual validation of candidates*.

For the *selection of candidates*, we may just take every colliding pair. However, we believe that counting the number of collision may help to rule out false positives.

As explained in Section 3, robust hashes for models are built as a concatenation of smaller hashes calculated from different (although possibly overlapping) model fragments extracted from the original model. Therefore, when collisions occur in different bands we can suspect that there are several parts of the involved MDE artefacts that are copied. This way, by counting the number of collisions we can provide as feedback a severity grade. From 0, meaning no plagiarism detected, to b collisions, meaning exact copies. We can then decide to focus only on pairs with more than n collisions to continue with the verification process. Note however that for this interpretation to be valid, we need to take into account the ratio between robust hash fragment size and the band size. When bands are larger than fragments sizes, we may miss some collisions (but on the other hand, the objective of larger bands is precisely to reduce the probability of collisions). Conversely, when bands are smaller than the fragment size we would need to discard (i.e., not count) intra-fragment collisions.

Once we have reduced our list of candidates of potential plagiarism we still need to

verify that the candidates are indeed suspected copies. For the *automatic validation of candidates*, we perform the actual pairwise comparison between the artefacts. This can be done at the robust hash level and at the original model level, depending of the available resources. In any of the two cases we need:

- a similarity (or distance) measure so that we can compare the copies.
- to determine threshold values w.r.t. this similarity measure for the *internal homogeneity* (i.e., how similar elements need to be for us to consider them copies) or conversely, the *external separation* (i.e., how different elements need to be so that we consider that they are not copies).

Finally, once we have obtained a list of models that are validated to be very similar, the final assessment implies the *manual validation of candidates*. This process should be probably followed by interviews with the involved students. Note that a final conclusion of plagiarism cannot be done fully automatically since there are many factors that may play a role in the automatic comparison (e.g., the degree of freedom in the original assignment) but our approach reduces the problem to a manageable process that provides, in our opinion, a good trade-off between precision and automation.

An extended version of the aforementioned process was followed for the validation of the results of this paper (see Section 4 for further details) which is summarized in the following:

(1) We passed the tool on all the available data (10 years of solutions);
(2) Automatic validation was used in order to eliminate lower similarity positives (which may be partial copies which we have decided to not classify as plagiarism);
(3) Author 1 manually checked that no high similarity pairs were discarded, pruned outliers (too small or empty files), selected a sample of suspicious pairs and gave them to Author 2 for further evaluation. The selection of pairs was done randomly on a by-assignment basis (i.e., we selected between 20% and 30% of pairs from each assignment so that all assignments were represented in the final sample);
(4) Author 2 (re)classified all sample pairs as suspicious and worth for further analysis. This validates the detection performed by the tool;
(5) Author 2 performed an in-deep analysis of the suspicious pairs considering not only the similarity values but also domain-specific hints (e.g., order of attributes in unnamed classes, comments, etc.) to finally classify (very conservatively, i.e., only clear plagiarism was taken into account) a number of suspicious pairs as plagiarism instances.

4. Evaluation

The authors of this paper have been responsible for preparing, teaching and evaluating MDE programs since many years. As such, we have collected a large number of models developed by our students over the years. At the time, we had to manually analyze all those models for plagiarism detection. From that experience we learned that telling the difference between very similar and arguably too similar models is very tedious and difficult to do manually by simple inspection.

To show the usefulness and applicability of the work presented in this paper, we now repeat the detection process in an automatic fashion thanks to our plagiarism tool for models.

Before proceeding to the actual evaluation, we present here a real use case consisting

in the analysis of two repositories of answers to MDE assignments in a MDE program organized since 2007.

4.1. Use Case: Model Transformation and Metamodels

The first repository of our use case contains answers to (meta)modeling exercises and the second repository answers to model transformation exercises. Students are required to use the Eclipse Modeling Framework (EMF) (Steinberg, Budinsky, Paternostro, & Merks, 2008) for the modeling exercises and the Atlanmod Transformation Language (ATL) (Jouault, Allilaire, Bézivin, & Kurtev, 2008) for the model transformation exercises. Note that according to Ciccozzi et al. (2018), EMF and ATL are the most popular technologies used in academia in order to teach modeling and model transformations respectively.

4.1.1. Modeling Assignments

EMF is a very popular modeling framework within academia and industry. It provides the means to create models and then automatically generate code and tooling (e.g., tree or form editors) for them. At the core of EMF is the Ecore metamodel, a metamodeling language that provides the basic concepts needed to describe domain models. Figure 3 shows the main concepts of Ecore. EPackages represent the root of the models, and contain all of its EClasses, this is, the entities of the domain being modeled. EClasses may contain a number of operations (for defining the behaviour of EClass) and EStructuralfeatures, that are either EAttributtes (basic value slots) or EReferences (links to instances of other EClasses).

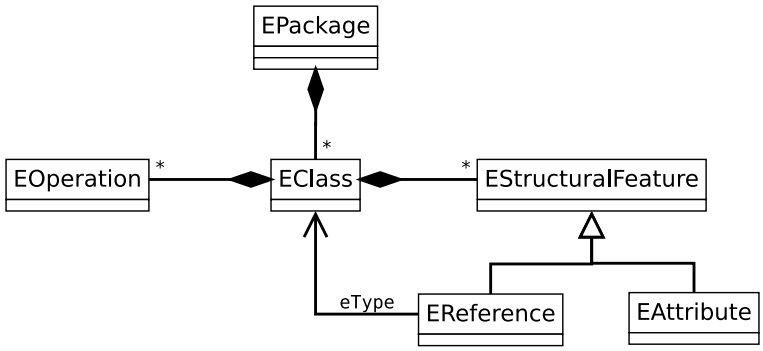


Figure 3. Ecore Language Excerpt

Once students have been introduced to modeling and to the EMF framework, they are asked to create a domain model using Ecore. They are provided with a general description of the task and with a set of examples to help them identify the main components of the domain model they have to built. In the following, we show a example of some of the instructions provided for an assignment consisting in creating a domain model (representing the RoverML modeling language metamodel) for Rovers:

1. *"RoverML is a modeling language for modeling rovers and the programs they execute. A rover model defines the roverâĂŹs topology, i.e., the individual components of a rover as well as the roverâĂŹs program, i.e., the commands which define its behaviour. Rovers are used in education (e.g., teaching kids programming), research (e.g., Mars*

rovers) and industrial applications (e.g., automated rovers in a warehouse transporting goods).”

2. *“A rover and its program are contained in a system. In such a system, multiple rovers and rover programs can exist. Example: The model in Figure 4 defines a rover and a assigned program.”*

3. *“Rovers have a name and consist of various components. Example: The Rover in Figure 4 is named Curiosity and has five components: motor, light, compass, position sensor and distance sensor.”*

4. *“Components of a rover have a unique name. There are two types of components: sensors and actuators. Sensors may be a GPS, a compass or a distance sensor. Possible actuators are motors and lights.*

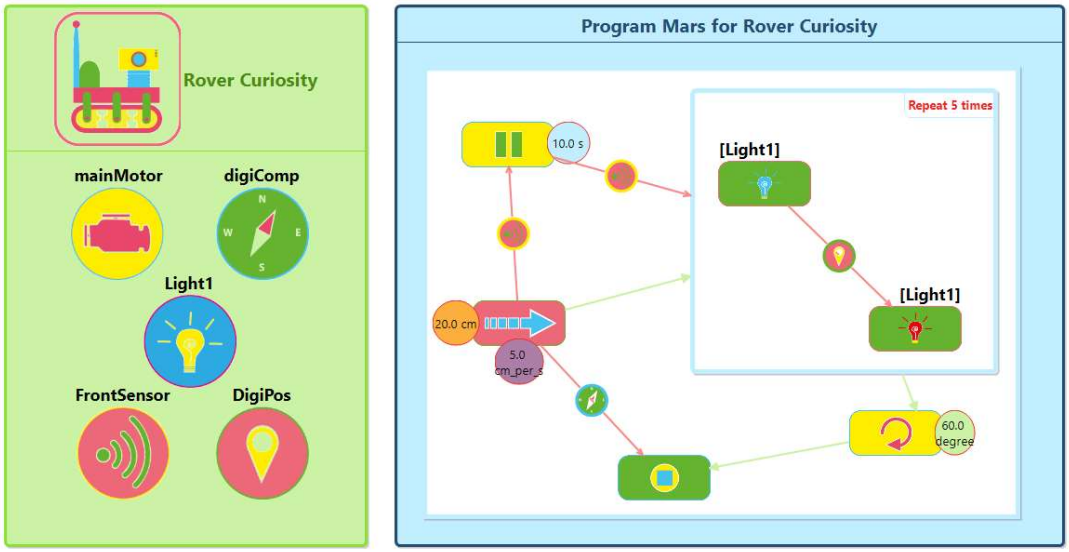


Figure 4. A Rover: Components and Program example given to the students as part of the assignment instructions

From the example above, it can be seen that both, vocabulary and structure, are given to the student limiting their freedom in choosing the concept names. Thus, students will produce very similar domain models.

4.1.2. Model Transformation Assignments

Model manipulation is a central activity in many model-based software engineering activities (Sendall & Kozaczynski, 2003). Model manipulations are usually implemented by means of model-to-model (M2M) transformations, and thus, the latter are often taught as part of MDE programs. A M2M transformation transforms a model M_a conforming to a metamodel MM_a into a model M_b conforming to a metamodel MM_b.

ATL is, due to its tooling and maturity, one of the most popular transformation languages among academic and industrial practitioners. It is, as mentioned above, the model transformation language most present in MDE programs. We show in top of Figure 5 an excerpt of the ATL abstract syntax (metamodel). An ATL transformation (module) is composed of a set of transformation rules and helpers. Each rule describes how (part of) the target model should be generated from (part of) the source model.

A helper can be seen as auxiliary function that enables the possibility of factorizing

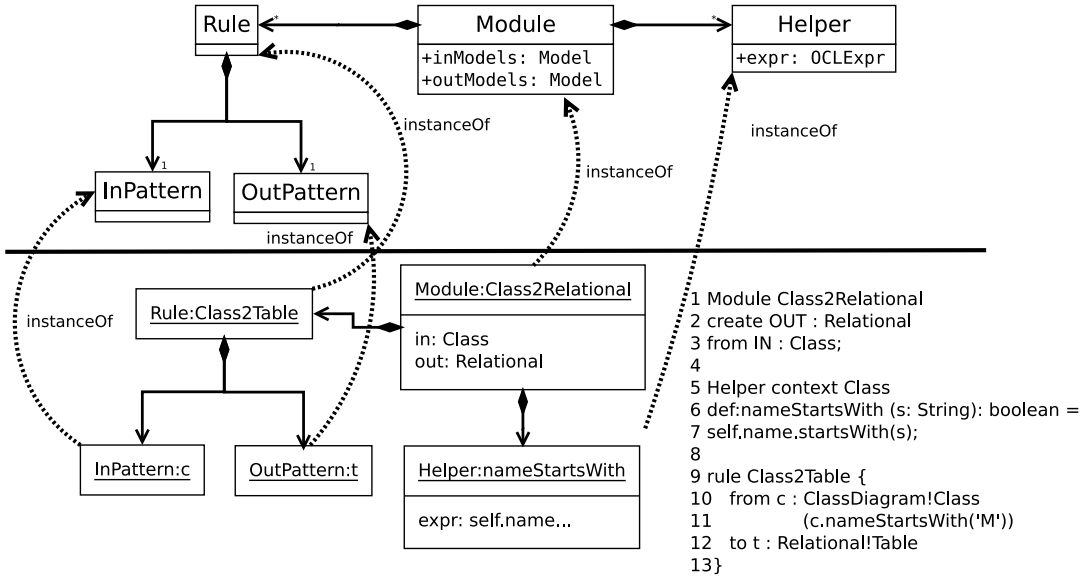


Figure 5. ATL Transformation Language Excerpt

ATL code used in different points of the transformation. In the transformation example depicted at the bottom of Figure 5, a transformation from *Class Diagram* to *Relational* models is defined (lines 1 to 3 specify name, input and output models). This transformations contains a rule called *Class2Table* (lines 9 to 13) that matches every *Class* which name starts with 'M' in the class diagram model to produce a *Table* in the output model. It does so by using a helper function that returns true if the name of a given model element starts by 'M' (lines 5 to 7). Rules are mainly composed of an input pattern (line 10 and 11) and an output pattern (line 12). The input pattern filters the subset of source model elements that are concerned by the rule. The output pattern details how the target model elements are created from the input ones. Each output pattern element can have several bindings that can be used to initialize the values of the elements in the target model. It is important to remark that while ATL transformations are specified by using a textual syntax, they are internally represented as models, and thus, can be manipulated as such.

In the following we show some of the instructions provided to the students in an exercise consisting in transforming RoverML models to UML models:

1. *"The goal of this assignment is to develop model-to-model transformations for the Rover Modeling Language (RoverML) using ATL [...]. In Part A of the assignment, you will develop an ATL transformation that translates RoverML models into Unified Modeling Language (UML) models. In addition, you will also use UML profiles to extend some elements of UML to allow transferring of some information from the RoverML models."*

2. *"Rover systems and rovers will be transformed into package diagrams and rover programs will be transformed into activity diagrams. UML 2 is used to specify, visualize, and document models of software systems, including their structure and design. It can be used for business modeling and modeling of other non-software systems too."*

3. *"Your transformation has to implement the following strategies for mapping RoverML model elements (source models) to UML model elements (target models):(see Table 1 for the list of mappings)"*

Table 1. RoverML to UML

RoverML	*UML*
RoverSystem	*Package. The packaged elements are the system's rovers and programs.*
Rover	*Package. The name of the package should be the name of the rover. The packaged elements are the components of the rover.*
RoverProgram	*Activity. The name of the activity should be the name of the program. The activity owns the node created from the program's block and belongs to the package of the rover the program has referenced in RoverML. The group of the activity is the block of the original program.*
Component	*Component. The name of the components should be the name of the original components. Apply the Actuator and Sensor stereotypes to the components accordingly. The last sensed value of sensors does not need to be transformed.*
Block	*Structured Activity Node. Its nodes are the commands of the block and its edges are the transitions of the block. If the block was a repeat block, apply the Repeat stereotype and set its count value.*
Transition	*Control flow. The control flow's name, source and target are taken from the original transition. Apply the Transition and TriggeredTransition stereotypes accordingly.*
Command	*The name is taken from the original command. Each command is mapped as an Opaque Action Node but a different stereotype for each command is available and should be applied accordingly.[. . .]*
Terminate	*Activity Final Node. Terminate is transformed differently than the other commands. Set the name of the node to the name of the original command and apply the Terminate stereotype.*

It can be seen that, as in the case of modeling assignment, precise instructions are given to the students. However, ATL and its embedded model navigation and query language, the Object Constraint Language (OCL) (Object Management Group, 2003), are somehow redundant languages. That is, the same result can be achieved in many different ways. In order to illustrate this, we show in Listing 1 and Listing 2 the excerpts of two answers produced by students. While they produce the same outputs for the same inputs, the two transformations use different language constructs, strategies, and internal organization. The first student uses one rule with several nested if statements in order to create the right outputs from a given *RoverML command*. Conversely, the second student uses a different rule for each *RoverML command*. As a consequence of this higher variability, and contrary to the modeling assignments, we should expect lower levels of similarity among student answers.

Listing 1 RoverML2UML from student X.

```
rule command2opaqueAction extends
    ↪abstract_command2opaqueAction  {
  from
    command: RML!Command (
      not command.oclIsTypeOf(RML!Repeat)
          ↪ and not
      command.oclIsTypeOf(RML!Terminate))
  to
    node: UML!OpaqueAction (
      name <- command.name
  )
  do {
    if (command.oclIsTypeOf(RML!Move)){
      [...]
    }else if (command.oclIsTypeOf(RML!
        ↪Rotate)){
      [...]
    }else if (command.oclIsTypeOf(RML!
        ↪Wait)){
    [...]
    }else if (command.oclIsTypeOf(RML!
        ↪SetLightColor)){
    [...}
    }else {
    -- do nothing
  }
  }
}
```

Listing 2 RoverML2UML from student Y.

```
rule Move2Action extends Command2Action {
    from
        command: RML!Move
    to
        action: UML!OpaqueAction
    do {
      [...]
    }
}
rule Rotate2Action extends Command2Action
    ↪ {
    from
        command: RML!Rotate
    to
        action: UML!OpaqueAction
    do {
      [...]
    }
}
rule Wait2Action extends Command2Action {
    from
        command: RML!Wait
    to
        action: UML!OpaqueAction
    do {
      [...]
    }
}
```

4.2. Evaluation setup & analysis

We devote this section to the analysis and experimental evaluation of our plagiarism detection approach for models. In order to do so we have developed a prototype implementation written in Java and based on the EMF API[2]. Using our prototype, and for each of the two types of assignments, i.e., modeling and model transformation assignments, we first determine the right parameters for the banding strategy by evaluating a number of samples and then we apply our approach with the previous configuration to all the repositories.

For each repository we: (*i*) show how many plagiarism candidates are detected; (*ii*) analyze each candidate pair to determine whether they constitute an actual instance of plagiarism.

We finish this section by providing performance and scalability evaluations.

4.2.1. Configuration of the number of bands

In order to tune our approach, this is, to determine the right number of bands for the LSH strategy, we need first to establish a desired similarity threshold. Then, we will aim to use a number of bands that make pairs of elements with similarity higher than the threshold to be positive (this is, collide in at least one band) and elements with similarity lower than the threshold to be negative. The calculation of the similarity for robust hashes is done using the *hamming similarity* measure for vectors, as shown in Equation 1. Basically, it counts the numbers of equal elements (*IdSim() returns 1 when elements are equal, 0 otherwise*) in the same position of the vectors and divides the result by the total number of elements.

$$HammingSimmilarity(s,t) = \frac{\sum_{i=1}^{n} IdSim(s[i],t[i])}{n} \quad (1)$$

[2]https://gitlab.com/smartine/RobustModelHashing/tree/master/LSHForModels

Table 2. Similarity thresholds per number of bands

b	1	2	4	5	8	10	20	25	40	50	100
t	1	0.99	0.97	0.96	0.92	0.89	0.74	0.66	0.47	0.37	0.1

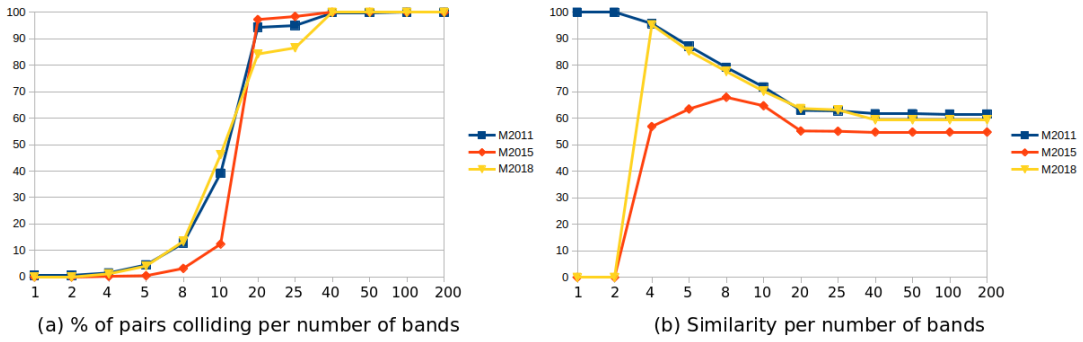


Figure 6. Model repository collisions and similarity per bands

Then, the Table 2 shows the theoretical approximative thresholds (as given by the formula $(1/b)^{1/r}$ (see Leskovec et al., 2014) for the different possible number of bands in a 200 element vector. From those values we can directly conclude that we are interested in numbers of bands lower than 25 and possibly bigger than 4. Users can directly work with these values and obtain good results. However, and in order to minimize the work left to be done by instructors, for both, model and transformation assignments, we analyze a sample obtained from our repositories to further reduce the choice for the number of bands. The obtained values may be used to analyse any similar repository, without further analysis.

4.2.2. Model Assignments

As discussed previously, we expect the models produced by the students in the modeling assignments to be very similar. In order to verify this assumption we calculate the average pairwise similarity of a number of model assignments repositories. Concretely we use repositories M2018, M2015, and M2011 (see /Appendix/ModelAssignments.ods in the web of the project for a detailed description of the repositories) for the calculation of the pairwise similarity. We obtain the following results: M2018=62, M2015=56, and M2011=62.

Then, we can proceed to verify if the theoretical values in Table 2 hold for our data and select the best number of bands for the plagiarism evaluation.

Band Configuration Analysis

Figure 6 summarizes the results of performing LSH to our three test repositories and calculating the number of collisions obtained with every band size.

In Figure 6(a) we see how the number of collisions obtained per number of bands is s-shaped. This is, very few pairs are obtained with small number of bands while almost all pairs collide with bigger number of bands, being the transition between both situations sharp. In Figure 6(b) we see what is the average similarity of the pairs that collided. It can be seen how it decreases as the number of bands grow until the general similarity average of the repository is reached. M2015 and M2018 do not have any colliding pairs for the lowest number of bands so the average similarity there is 0. M2011 shows a similarity of (near) 100% is reached when doing LSH with 1 and 2 bands, meaning

an exact copy is detected. Interestingly, M2015 grows before decreasing, meaning that collision(s) happened with very low similarity. This may constitute a false positive or a partial copy.

Summarizing, from the aforementioned tests we can conclude that the ideal number of bands for classifying modeling assignments is to be chosen between 4 and 8.

Repository Evaluation

Being that there is a clear decrease of the similarity among the collision when using 8 bands and that we are interested in highly similar pairs, we choose 4 as our number b. Thus, we applied LSH to all the available repositories by using $b = 4$ and $r = 40$. The results are summarized in Table 3, where we show for each of the repositories the number of plagiarism candidate pairs (i.e., collisions), the average similarity among these candidates and finally, the number of candidates that are considered final candidates after the automatic verification step that removes pairs with similarity values below a threshold (we set this threshold at 85, that is above the average similarity obtained when using the extreme of the useful range for the number b, this is 8). We show the results of this evaluation in Table 3.

In most cases a few candidates of plagiarism are found. From them, very few are ruled out as false positives after automatic verification. The case of M2012 is interesting, as we obtain a number of candidate pairs much higher than average. After manual verification we saw two things: 1) students were asked to build two different models; and 2) these domain models are very small (i.e., they less than 10 elements). This perfectly explains the obtained results, as this assignment does not offer enough variability for a plagiarism evaluation. This finding is in line with the conclusions of Rosales et al. (2008), stating that programming assignments composed of small subroutines can not be effectively evaluated for plagiarism detection.

Finally, we had two of the authors of this paper (with experience as instructors in MDE courses) inspect a sample of the modeling assignments plagiarism candidates. While instructors used a very conservative criteria (e.g., they only accepted as plagiarism pairs with very high similarity and suspiciously equal choices in diverse parts of the model), they both found a number of previously undetected plagiarism instances, what evidences the usefulness of our approach.

Additionally, we verified (for a selected number of repositories) that no false negatives, this is, pairs with high similarity (similarity equal or higher than the average for the band number) that did not collide, were obtained.

Table 3. Modeling Assignments Plagiarism Tests

Repository	Raw Candidates	Similarity	Candidates above 85% threshold
M2007	2	94.50	2
M2008	0	0.00	0
M2009	10	97.80	10
M2010	10	92.45	9
M2011	9	95.72	9
M2012	81	95.64	81
M2013	4	100.00	4
M2014	7	90.20	6
M2015	1	56.99	0
M2016	0	0.00	0
M2017	22	91.95	20
M2018	2	95.25	2

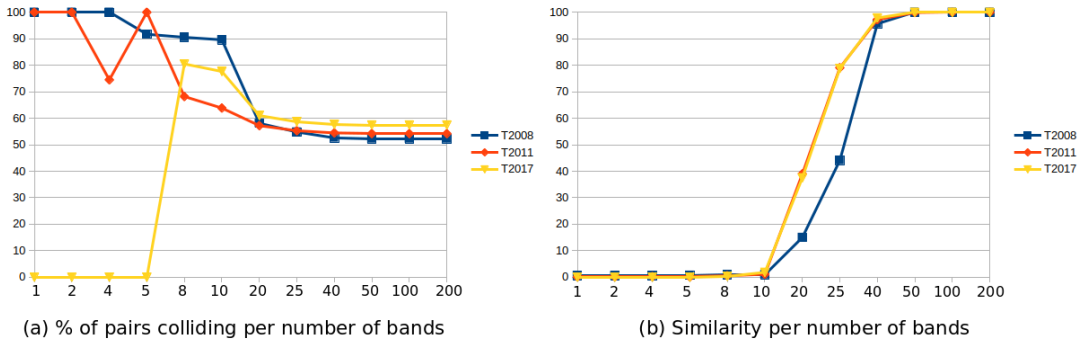


Figure 7. Transformation repository collisions and similarity per bands

4.2.3. Model Transformation Assignments

As already mentioned, model transformation assignments differ from modeling ones. They are completed by using a transformation language that gives students more freedom to obtain the correct result. Moreover, models representing transformations are much bigger (note that robust hashes are calculated with different parameters w.r.t. the modeling assignments, as these parameters are determined as a factor of the average model size). Therefore, we repeat the analysis and evaluation process we performed for the modeling assignments in order to determine the right number of bands.

Band Configuration Analysis

In this case, we use repositories T2017, T2011, and T2008 (see /Appendix/TransformationAssignments.ods in the web of the project for a detailed description of these repositories)) for the configuration evaluation. We start by calculating their pairwise similarity, obtaining the following results: T2017=57, T2011=54, and T2008=52. As expected, the average similarity is lower than the similarity of repositories containing model assignments.

Then, we perform the calculation of the number of collisions obtained for each of our test repositories (T2017, T2011, and T2008) with all the possible number of bands. Figure 7 summarizes the results obtained by performing the LSH to our three test repositories. From these results we can conclude that the ideal number of bands for classifying transformation assignments is to be chosen between 8 and 10.

Repository Evaluation

We apply LSH to all the available transformation repositories by using $b = 8$ and $r = 25$. The results are summarized in Table 4, where we show for each of the repositories the number of plagiarism candidate pairs (i.e., collisions), the average similarity among these candidates and finally, the number of candidates that are considered final candidates after the automatic verification step that removes pairs with similarity values below a threshold (we set this threshold at 80, that is above the average similarity obtained when using the extreme of the useful range for the number b, this is 20). In most of the cases a few candidates of plagiarism are found. Contrary to the case of model assignments, a significant number of candidate pairs are eliminated after automatic verification. Even when model transformation languages offer a lot of variability, some of the rules in a MDE assignment may easily end up being exactly equal among different student answers, by partial copy (due to available tooling partially copying transformation assignments is much easier than partially copying models) or induced

by the simplicity of the assignment, even when the average similarity of the full model is low. Arguably, a partial copy would not be considered an instance of plagiarism from an instructor, and thus, we believe that setting a high threshold for the automatic verification is the best choice.

As with the modeling assignments, we verified (for a selected number of repositories) that no false negatives were obtained. Then, we have two instructors inspect a sample of the plagiarism candidate pairs. Previously undetected plagiarism instanced were found as well.

Table 4. Transformation Assignments Plagiarism Tests

Repository	Raw Candidates	Similarity	Candidates above 80% threshold
T2007	13	72.00	4
T2008	9	90.55	7
T2009	3	90.80	2
T2010	3	71.30	1
T2011	5	68.20	1
T2012	4	59.75	0
T2013	4	92.37	4
T2014	6	64.60	0
T2015	9	81.83	5
T2016	5	62.30	0
T2017	1	80.50	1
T2018	18	76.25	4

4.2.4. Performance Evaluation

Performance and scalability are two very important aspects of any plagiarism detection tool as a slow tool that only works with very small repositories will not be adopted by instructors. Moreover, the re-utilization of previous year assignments by instructors together with the proliferation of public repositories and social coding platforms, hints to an steady increase in the size of the repositories that will need to be analysed in order to deal with plagiarism.

Therefore, we need to evaluate the performance and scalability of our approach w.r.t. the existing alternatives. These alternatives are: 1) the direct pairwise comparison of the models with a model comparison tool; 2) the direct pairwise comparison of the robust hashes obtained from the models.

Figure 8 shows the times (we used an IntelÂő CoreâĎć i5-6200U CPU @ 2.30GHz x 4 cores, running Ubuntu 16.04) taken for the analysis of repositories of increasing size (from 10 to 400 models) by using model comparison, robust hash comparison and LSH. The repositories are composed of model assignment answers, this is, medium to small size models (40 to 100 classes). It can be seen that the direct comparison of models (We use EMFCompare (Toulmé, 2006) for the model comparison, a widespread tool for the the differencing and matching of EMF models) performs poorly as the repository grows, taking roughly 40 minutes for the analysis of a repository of 200 models. Conversely, the pairwise comparison of robust hash and LSH are orders of magnitude faster.

We, thus, focus our attention on comparing the two more promising approaches: pairwise comparison of robust hashes versus LSH.

Figure 9 shows the results of this experiment. It can be seen that the LSH performs much better than the pairwise comparison of robust hashes. More importantly, LSH

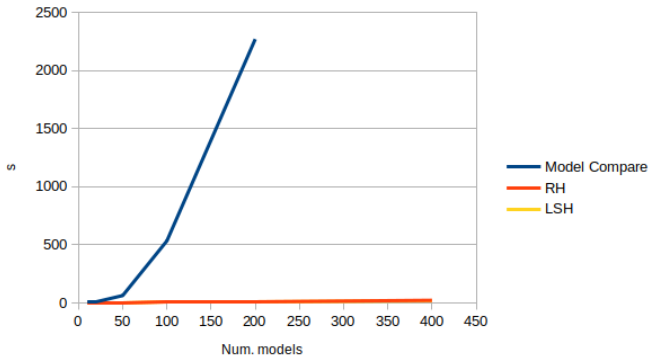


Figure 8. Compared performance of Model Comparison, RH comparison and LSH

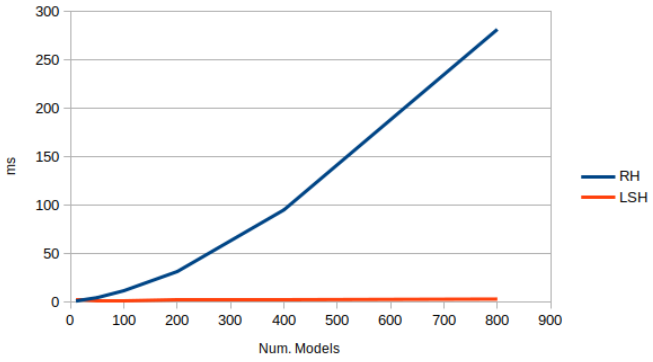


Figure 9. Pairwise Robust Hash Comparison vs LSH

scales very well as the model repository grows, showing an almost linear behaviour.

4.2.5. Threats to Validity

The two main types of validity threats to the evaluation and conclusions of our approach are external and internal validity.

Internal Validity

The validity of the conclusions obtained by our experiments may be affected mostly by the ambiguous character of the term "plagiarism". In effect, the classification of a piece of work as an instance of plagiarism depend often on domain specific thresholds for automatic verification and the subjective opinion of the experts in a manual verification. We mitigate the effects of these threats by (*i*) evaluating the average similarity of the MDE assignments, so that we can better approximate a good similarity threshold for the automatic verification; (*ii*) having different authors independently perform the manual verification as we explain in Section 4, so that we rule out individual biases.

External Validity

We use for our evaluation a set of repositories composed of more than 10 years of answers to MDE assignments. Regarding these repositories, two threats must be taken into account. First, the technologies used for the assignments; and secondly, the origin of the repositories.

W.r.t. the first threat, the repositories were composed of modeling assignment answers created by using the EMF modeling framework and model transformations assignment answers created by using the ATL transformation language. Although Ciccozzi et al. (2018) show that these technologies are the most popular among MDE programs, our approach and evaluation should be extended to other technologies in order to rule out effects of technology in the rates of plagiarism. As for the second threat, all repositories come from an MDE program taught in the same institution. In this sense, MDE assignments from other institutions may be required in order to generalize the conclusion of this work.

5. Related Work

Computing assignments plagiarism is an ongoing concern for academic institutions and thus, it remains an open subject in the computer education research community.

Plagiarism of software artifacts has been studied at the programming level in both general (see Ďuračík et al., 2017; Liu, Chen, Han, & Yu, 2006; Narayanan & Simi, 2012) and education scenarios (Bejarano et al., 2015; Inoue & Wada, 2012; Joy & Luck, 1999; Lancaster & Culwin, 2004; Rosales et al., 2008). The aforementioned approaches are not directly re-usable for MDE artifacts (i.e., structured data graphs) as they are tailored to work with source code (i.e., text). To the best of our knowledge, our work is the first specially tailored to the detection of plagiarism in modeling (analysis, design,...) courses and MDE-related education assignments.

At a technical level, the work that most resembles our approach is the one contributed by Cochez (2014), where the author uses minhash and LSH in order to match equivalent terms in different ontologies. Nevertheless, that approach works at the individual class / term level and therefore cannot be used for plagiarism detection at the global model level (e.g., structure is not taken into account).

On a broader perspective, model clone, model comparison and model matching approaches may be regarded as related to the problem of plagiarism detection. However, they are generally designed with a different purpose: finding similar intra-model (and to a lesser extent inter-model) fragments generated by a legitimate copy and paste reuse. Approaches for model clone detection have been proposed on UML models (Störrle, 2015), graphs (Pham, Nguyen, Nguyen, Al-Kofahi, & Nguyen, 2009), Simulink models (Deissenboeck, Hummel, Juergens, Pfaehler, & Schaetz, 2010) or rule-based model transformations (Strüber, Acreţoaie, & Plöger, 2017) while other proposals target general model comparison (Brun & Pierantonio, 2008), model alignment (Falleri, Huchard, Lafourcade, & Nebut, 2008; Kolovos, 2009) or model versioning (Constant, 2012). In theory, all these works could be systematically executed on all combinations of models in a repository to find suspicious pairs of models but in practice this is not really feasible. They rely in the execution of a large number of matching and comparison operations (they are basically refinements of the graph theoretic problem of locating isomorphic connected sub-graphs of certain size, which is know to be NP-complete) that will not scale to the analysis of large model repositories.

Model clustering approaches may be considered similar to the problem of plagiarism detection as well. In this area, Kinneer and Herzig (2018) provide and test different similarity measures for the clustering of domain specific models (space mission architecture models). Similarly, the work presented by Dumas, García-Bañuelos, La Rosa, and Uba (2013); La Rosa et al. (2015) focus on the clustering of Business Process Models. Clarisó and Cabot (2018) rely on Graph Kernels for the clustering phase. Nev-

ertheless, and contrary to our approach, clustering approaches work out-of-the-box only for very specific and predefined types of models or, otherwise, require to define a specific *distance* function for the problem at hand and proceed with a pairwise comparison.

Finally, Babur, Cleophas, and van den Brand (2019) follow a different strategy, based on the use of n-grams and natural language processing (NLP) techniques (in the NLP field, an n-gram is a contiguous sequence of n items from a given sample of text or speech (Cavnar, Trenkle, et al., 1994)). Basically it builds a weighted term incidence matrix (terms are more or less complex n-grams to be extracted and matched in the models under comparison) and then calculates the similarity between vectors in this matrix or feed it to clustering algorithms. We argue that our approach scales better as, again, avoids matching and pairwise comparisons. However, we believe their approach may be an interesting complementary technique in our final phase (when we do proceed with individual comparisons for the final filtering) as it is able to point to the parts of the artefact that it is presumably copied.

6. Conclusions and Future Work

We have presented a new plagiarism detection mechanism for MDE course assignments. Our approach is based on the adaptation to the MDE domain of the Locality Sensitive Hashing technique.

We evaluated the feasibility, usefulness and efficiency of our approach on two real use cases featuring 10 years of assignments about modeling and model transformations pertaining to a MDE course taught between 2007 and 2018. We showed that: (*i*) our approach succeeds in massively reducing the time it takes to assess potential plagiarisms by preselecting the suspicious candidates; (*ii*) our approach scales much better than existing alternatives, such as using model comparison tools; (*iii*) it was able to detect (previously undetected) instances of plagiarism that did exist in the use case repository. We can conclude that plagiarism detection must be integrated into the toolset and activities of instructors in order to correctly evaluate students and the outputs of MDE programs and that our approach succeeds in doing so.

As future work we intend to extend the present work by exploring a number of different research lines. To begin with, we are interested in replicating our empirical evaluation on MDE programs taught in different institutions, so that we can verify how different kinds of assignments, related tools and instructions affect the plagiarism detection rates and update the tool accordingly. This will also help us to come up with good default values and heuristics for the tool configuration based on the types of models to evaluate (UML models, business process models, database diagrams,...), which would also simplify the adoption of our tool by other educators. Finally, we are interested in evaluating the long-term impact of our approach in the MDE teaching domain by looking at how both instructors and students adapt to the existence of the tool. For instance, we consider the following questions worth to be investigated: Does the tool facilitate instructors to reuse previous assignments with less concerns on students copying them? Do students attempt to defeat the tool by being more creative with the plagiarisms or is the tool a barrier that forces them to actually do the work by themselves?

References

Babur, Ö., Cleophas, L., & van den Brand, M. (2019). Metamodel Clone Detection with SAMOS. *Journal of Computer Languages*, *51*, 57 - 74.

Bejarano, A. M., García, L. E., & Zurek, E. E. (2015). Detection of source code similitude in academic environments. *Computer Applications in Engineering Education*, *23*(1), 13–22.

Bézivin, J. (2005). On the unification power of models. *Software & Systems Modeling*, *4*(2), 171–188.

Blum, S. D. (2009). *My word!: Plagiarism and college culture.* Cornell University Press.

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice.* Morgan & Claypool Publishers.

Broder, A. Z. (1997). On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences* (pp. 21–29).

Brosch, P., Kappel, G., Seidl, M., & Wimmer, M. (2009). Teaching model engineering in the large. In *Educators' Symposium @ Models 2009.*

Brun, C., & Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, *9*(2), 29–34.

Cavnar, W. B., Trenkle, J. M., et al. (1994). N-gram-based text categorization. In *Proceedings of the 3rd Symposium on Document Analysis and Information Retrieval (SDAIR).*

Ciccozzi, F., Famelis, M., Kappel, G., Lambers, L., Mosser, S., Paige, R. F., ... others (2018). How do we teach modelling and model-driven engineering?: a survey. In *Companion Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 122–129).

Clarisó, R., & Cabot, J. (2018). Applying graph kernels to model-driven engineering problems. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis (MASES@ASE)* (pp. 1–5).

Cochez, M. (2014). Locality-sensitive hashing for massive string-based ontology matching. In *Proceedings of the IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)* (pp. 134–140).

Constant, O. (2012). *EMF Diff/Merge.* Website. (Online available: `https://wiki.eclipse.org/EMF_DiffMerge`)

Deissenboeck, F., Hummel, B., Juergens, E., Pfaehler, M., & Schaetz, B. (2010). Model clone detection in practice. In *Proceedings of the 4th International Workshop on Software Clones* (pp. 57–64).

Dumas, M., García-Bañuelos, L., La Rosa, M., & Uba, R. (2013). Fast detection of exact clones in business process model repositories. *Information Systems*, *38*(4), 619–633.

Ďuračík, M., Kršák, E., & Hrkút, P. (2017). Current trends in source code analysis, plagiarism detection and issues of analysis big datasets. *Procedia engineering*, *192*, 136–141.

Falleri, J.-R., Huchard, M., Lafourcade, M., & Nebut, C. (2008). Metamodel matching for automatic model transformation generation. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems* (pp. 326–340).

Fridrich, J., & Goljan, M. (2000). Robust hash functions for digital watermarking. In *Proceedings of the International Conference on Information Technology: Coding and Computing* (pp. 178–183).

Indyk, P., & Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th ACM Symposium on Theory of Computing* (pp. 604–613).

Inoue, U., & Wada, S. (2012). Detecting plagiarisms in elementary programming courses. In *Proceedings of the 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)* (pp. 2308–2312).

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: a Model Transformation Tool. *Science of Computer Programming*, *72*, 31–39.

Joy, M., & Luck, M. (1999). Plagiarism in programming assignments. *IEEE Transactions on Education*, *42*(2), 129–133.

Kinneer, C., & Herzig, S. J. (2018). Dissimilarity measures for clustering space mission architectures. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 392–402).

Kolovos, D. S. (2009). Establishing correspondences between models with the Epsilon Comparison Language. In *Proceedings of the European Conference on Model Driven Architecture-Foundations and Applications* (pp. 146–157).

Lancaster, T., & Culwin, F. (2004). A comparison of source code plagiarism detection engines. *Computer Science Education*, *14*(2), 101–112.

La Rosa, M., Dumas, M., Ekanayake, C. C., García-Bañuelos, L., Recker, J., & ter Hofstede, A. H. (2015). Detecting approximate clones in business process model repositories. *Information Systems*, *49*, 102–125.

Lee, S.-H., & Kwon, K.-R. (2012). Robust 3D mesh model hashing based on feature object. *Digital Signal Processing*, *22*(5), 744–759.

Leskovec, J., Rajaraman, A., & Ullman, J. D. (2014). *Mining of massive datasets*. Cambridge University Press.

Liu, C., Chen, C., Han, J., & Yu, P. S. (2006). GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 872–881).

Martínez, S., Gérard, S., & Cabot, J. (2018). Robust hashing for models. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 312–322).

Narayanan, S., & Simi, S. (2012). Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In *Proceedings of the 7th International Conference on Computer Science & Education (ICCSE)* (pp. 1065–1068).

Object Management Group. (2003). OCL Specification 2.0. *OMG Adopted Specification (ptc/03-10-14)*.

Pham, N. H., Nguyen, H. A., Nguyen, T. T., Al-Kofahi, J. M., & Nguyen, T. N. (2009). Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)* (pp. 276–286).

Reddy, R., France, R., Ghosh, S., Fleurey, F., & Baudry, B. (2005). Model composition-a signature-based approach. In *Aspect Oriented Modeling (AOM) Workshop*.

Rosales, F., García, A., Rodríguez, S., Pedraza, J. L., Méndez, R., & Nieto, M. M. (2008). Detection of plagiarism in programming assignments. *IEEE Transactions on Education*, *51*(2), 174–183.

Sendall, S., & Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE Software*, *20*(5), 42–45.

Starovoytova, D., & Namango, S. S. (2016). Viewpoint of undergraduate engineering students on plagiarism. *Journal of Education and Practice*, *7*(31), 48–65.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse Modeling Framework (2nd edition) (The Eclipse Series)*. Addison-Wesley Professional.

Steinebach, M., Klöckner, P., Reimers, N., Wienand, D., & Wolf, P. (2013). Robust Hash Algorithms for Text. In *Proceedings of the 14th IFIP TC 6/TC 11 International Conference on Communications and Multimedia Security (CMS)* (pp. 135–144). Springer.

Störrle, H. (2015). Effective and efficient model clone detection. In *Software, Services, and Systems* (pp. 440–457). Springer.

Strüber, D., Acreţoaie, V., & Plöger, J. (2017). Model clone detection for rule-based model transformation languages. *Software & Systems Modeling*, 1–22.

Toulmé, A. (2006). Presentation of EMF compare utility. In *Eclipse Modeling Symposium* (pp. 1–8).

Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE Software*, *31*(3), 79–85.