Noname manuscript No.
(will be inserted by the editor)

Efficient Model Similarity Estimation with Robust Hashing

Salvador Martínez · Sébastien Gérard · Jordi Cabot

Received: 27 April 2020 / Accepted: 19 July 2021

Abstract As Model-Driven Engineering (MDE) is increasingly adopted in complex industrial scenarios, modeling artefacts become a key and strategic asset for companies. As such, any MDE ecosystem must provide mechanisms to protect and exploit them. Current approaches depend on the calculation of the relative similarity among pairs of models. Unfortunately, model similarity calculation mechanisms are computationally expensive which prevents their use in large repositories or very large models.

In this sense, this paper explores the adaptation of the Robust Hashing technique to the MDE domain as an efficient estimation method for model similarity. Indeed, robust hashing algorithms (i.e. hashing algorithms that generate similar outputs from similar input data), have proved useful as a key building block in intellectual property protection, authenticity assessment and fast comparison and retrieval solutions for different application domains. We present a detailed method for the generation of robust hashes for different types of models. Our approach is based on the translation to the MDE domain of diverse techniques such as summary extraction, minhash generation and locality sensitive hash function families, originally developed for the comparison and classification of large datasets. We validate our approach with a prototype implementation and show that: 1) our approach can deal with any graph-based model representation; 2) a strong correlation exists between the similarity calculated directly on the robust hashes and a distance metric calculated over the original models; 3) our approach scales well on large models and greatly reduces the time required to find similar models in large repositories.

Keywords Model-driven Engineering · Locality Sensitive Hashing · Near-Similar Search · Robust Hashing Comparison

Salvador Martínez
Lab-STICC, IMT Atlantique.
Brest, France
E-mail: salvador.martinez@imt-atlantique.fr

Sébastien Gérard
CEAList
Paris-Saclay, France
E-mail: sebastien.gerard@cea.fr

Jordi Cabot
ICREA - UOC
Barcelona, Spain
E-mail: jordi.cabot@icrea.cat

1 Introduction

The increased adoption of model-driven engineering (MDE) tools and techniques in complex industrial environments (e.g., MDE for cyber-physical systems, Internet of Things or Industry 4.0.) highlights the value of modeling artefacts. Indeed, in such complex scenarios, valuable (domain-specific) knowledge is produced, exchanged and consumed in the form of a growing number and diversity of models and metamodels.

This introduces as a new requirement in MDE the need to both protect and take full advantage of these valuable assets. As such, we believe that the MDE ecosystem needs to provide the means to 1) protect the MDE assets' owners from losing their intellectual property 2) protect MDE assets' users from malicious tampering aimed at damaging or putting at risk the assets 3) efficiently store, retrieve and compare MDE assets, so that reuse is maximized.

In order to deal with the aforementioned requirements, in this paper we explore the adaptation of the concept of *robust hashing* to the MDE domain. Indeed, robust hashing algorithms have proven useful as a key building block for providing intellectual property protection, authenticity assessment and fast comparison and retrieval solutions in different application domains such as digital images [1], 3D models [2] or text documents [3]. Contrary to cryptographic hash algorithms, such as MD5 [4] or SHA1 [5], where slightly different inputs produce very different outputs due to the avalanche effect [6], robust hashing algorithms (often called perceptual hashing algorithms) produce the same or very similar hashes for similar inputs. Moreover, they are capable of resisting attacks (i.e., modifications) that change non-essential properties of the asset.

Therefore, we propose a novel robust hashing algorithm adapted to the characteristics of MDE technical space. Our algorithm starts by extracting multiple overlapping fragments from the model to be hashed, so that model elements are characterized not only by their contents (e.g., attributes and operations) but also by their relative position w.r.t. other model elements. Fragments are then translated to textual set summaries so that we can apply to them the *min-wise independent permutations locality sensitive hashing scheme (minhash)* [7], that reduces large summary sets to small and robust hash signatures. Finally hash signatures are manipulated and combined to obtain the final robust hash of a given model.

We demonstrate the feasibility of our approach by a prototype implementation and its corresponding evaluation w.r.t. two key properties: robustness (i.e., resistance to mutations and thus to false negatives) and discrimination (absence of false positives).

This paper significantly extends our previous work on this topic [8] by 1) generalizing the approach to make it useful for a broader range of models; 2) adding a new section describing how to implement the approach on top of the Eclipse Modeling Framework [9] in a way that it is able to deal with model instances conforming to any metamodel; 3) presenting a boundary analysis of the parameters that guide the approach. This analysis shows a strong correlation with a metric calculated on the original models and improves the discrimination and robustness of the approach with respect to previous work (see [8]); 4) contributing a scalability evaluation demonstrating that our approach can deal with big repositories and very large models.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts of a robust hashing scheme and Section 3 their adaptation to MDE. Section 4 provides details about our hashing algorithms and its building blocks. Section 5 gives details about the prototype implementation and discusses how it can be adapted to different types of models. Analysis and evaluation of our robust hashing approach are provided in Section 6, followed by a discussion of application scenarios in Section 7. Section 8 discusses related work. We present conclusions and future work in Section 9. Finally, we describe in the Appendix additional experiments to validate our approach.

2 Background On Robust Hashing

In this section we introduce the basic concepts and properties related to robust hashing.

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 was introduced, notably in the domain of digital images [1] but also in other domains such as those of 3D model meshes [2] and textual documents [3]. A robust hash is a hash that can resist a certain type and/or number of data manipulations, i.e. 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 meshes 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.

The main building blocks of a robust hashing scheme are depicted in Figure 1. Basically, a robust hash algorithm starts by extracting key features from the data to hash, then groups and hashes them to finally use aggregation/selection/compression methods to obtain the final data hash. We describe each of these building blocks in the following. Note that these building blocks may be decomposed in several substeps.

– *Feature extraction.* Core features of the data to be protected are extracted (e.g., for images this could be histogram information, wavelet transform information, etc.). The key idea of this feature extraction

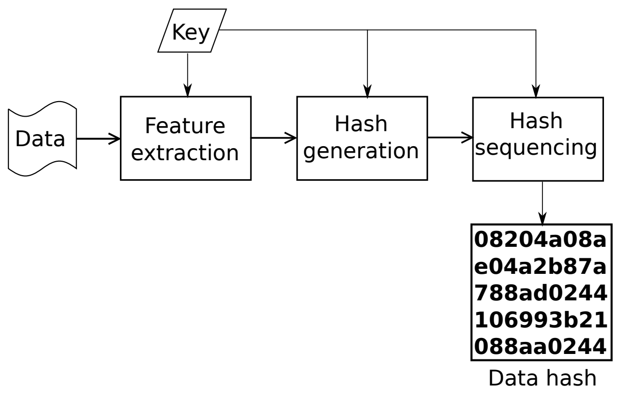


Fig. 1: Robust Hashing Generation Algorithm

step is to focus on those characteristics that give the data its main *meaning*, so that changes in less important features do not lead to very different hashes. This step normally divides the input data into several different data groups, then applies a randomization step to scramble the data groups.

– *Hash Generation.* The core features extracted in the previous step are then further manipulated and finally hashed using different mechanisms (that can include cryptographic hashes). As with the previous step, randomization operations are applied to the results to further scramble the hash.
– *Hash Sequencing.* The hashes generated from features in the previous step are transformed into a final hash. This step may include different operations with different objectives such as compressing the hash or augmenting its robustness. Among the typical operations that we find in this block we have: 1) quantization; and 2) error correction codes.
– *Key.* Unlike standard cryptographic hash algorithms, robust hash algorithms are by construction vulnerable to *pre-image attacks*, notably to *second pre-image* attacks. Knowing the hash algorithm, it is not difficult to find an input that leads to a given hash. And given a data input and a hash it is not difficult to build a second input with the same or a very similar hash. To alleviate this problem, the previously described steps use randomization operations. As we need the randomization to be *reproducible* so that we can do hash comparisons, a secret key is used as a seed to the pseudo-random number generator used at various stages of the robust hashing algorithms. Given the same data, if a different key is used, the resulting hash will be statistically independent and thus completely different. The key is kept secret, and so, the hash value of a given piece of data cannot be computed or verified by an unauthorized party.

2.2 Robust Hashing Properties and Requirements

We have introduced the concept of robust hashing and its composing building blocks. Here we present two properties that characterize robust hashing algorithms: *Robustness* and *Discrimination*. We define them below:

– *Robustness* refers to the capacity of a robust hashing algorithm to resist to data distortion due to 1) its normal manipulation; or 2) intentional attacks aimed at hiding the similarity between the data. In other words, it refers to the capacity of the robust hash algorithm to avoid producing false negatives in the face of certain data manipulations that do not change the core features of the data.
– *Discrimination* refers to the ability of a robust hashing algorithm to classify as different two (sufficiently) different data inputs. This is, to be usable, a robust hash algorithm needs to avoid producing false positives.

Summarizing, when designing a robust hash algorithm, both robustness and discrimination need to be assured. There is a trade-off between both properties: the more robust, the more prone to false positives and vice versa.

As extra requirements for a robust hashing scheme we have the size of the hash and its internal organization. Hashes must have a regular size and their features must be scrambled following a predictable algorithm. Otherwise, the hash comparison for determining authenticity of the data will not be feasible (calculating similarity and distance of irregular data is much more difficult and often extremely slow). Moreover, note that for certain applications, the *detection* of similar hashes will be a probabilistic process. This is, we may not find exact the same hash but a *very* similar one. Thus, threshold criteria would have to be defined to discriminate between similar and non similar hashes.

3 Requirements for a Robust Hashing Solution for Models

This section discusses how the concept of robust hashing and its properties can be adapted to properly enable the hashing of models. The next sections describe how we take these requirements into account in our proposal for a robust hashing algorithm for models.

In an abstract, formal way, we can see models as attributed graphs $G = \langle V, E, a \rangle$ (we adapt here the formalization and notation described in [10]) where vertices are the model elements, edges $e : V \to V \in E$

are the relations between model elements and attributes $a : V \times \Sigma^* \rightarrow value$, a function from the Cartesian product of vertex and attribute identifiers (Σ being a set of alphanumeric characters) to concrete values. Additionally, models can be regarded as structured data (the structure is defined by the metamodel the model conforms to). Similarly, metamodels conform to metametamodels and, as such, they can be regarded (and manipulated) as models as well; we will use indistinctly the term model to refer to both unless disambiguation is necessary. In fact, all MDE artefacts, that can be represented as models themselves [11], will benefit from our approach .

In this scenario, and based on the definitions from the previous section, we need to answer the two following questions in order to design a robust hashing algorithm for models: 1) what is the information of a model to be regarded as essential?; 2) what are the model modifications or attacks that need to be resisted by our algorithm?

3.1 Essential features

Model elements include individual data (modeled as the attribute function defined above) but are also related to a number of other model elements via their references (edges). 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 but that, by chance, share a similar structure, could be regarded as equivalent (note that for simplicity, we model all model element data as attributes, including introspection information such as types).

3.2 Type of attacks

There are many types of model manipulations and transformations. Some of them should not have any impact in the resulting hash as they do not change the model itself but just its representation. Others do change the model content and therefore should only be tolerated to a certain degree before the model ends up generating a different hash.

Concretely:

– Changes in the way a model is serialized should not have any impact on the resulting hash. The same model stored as an XMI document, as table rows in a relational database or as a graph object in a graph database should always result in the same hash. Note that this *feature* is already supported in many other model management tasks that work at the conceptual level and abstract the serialization specificities.
– Mutations to the model content (e.g., for UML structural models, changes to classes, attributes, references, operations) should be resisted but only to a certain degree. A structural model containing just a few new classes and references should be still considered the same (or a derived) model (as per the robustness property) with respect the original one. This resistance should not be that strong that models that become different enough to be perceived as so by a human are classified as equal (as for the discrimination property).

4 Approach

We now present our proposal for a robust hashing mechanism for models. Figure 2 depicts its main steps. It starts by creating m fragments of size s from the model to hash. These model fragments are then represented as text summaries so that we can apply on them the *minhash* hashing technique to obtain m small model fragment signatures. Once the model signatures have been generated, the next step consists in dividing and *re-hashing* them to perform a content-based classification in a number of buckets. Next, the global model hash is created by taking n elements from key-selected buckets. Finally, the hash is compressed using a scalar quantization step.

We devote the rest of this section to a detailed description of the rationale and functioning of each of these steps. Note that in order to illustrate the discussion we use as a concrete target example a specific type of model: UML structural models. Nevertheless, our approach is designed to work for any kind of model.

4.1 Model Fragmentation

As a first step we create a number of fragments from the model. This allows us 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.

Once we have determined the need for the extraction of model fragments we need to decide: 1) how our fragments are created and 2) how many of these fragments are needed. The answer to these two questions is determined by the purpose of our method, that is

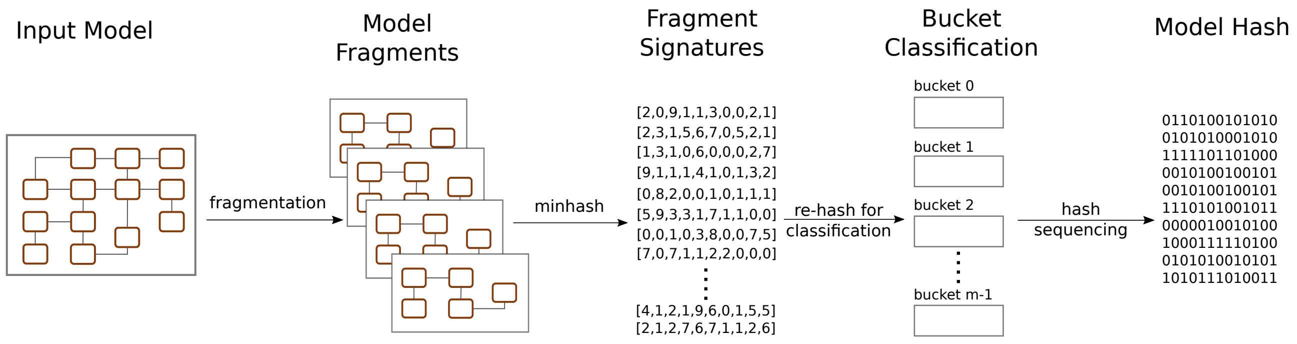


Fig. 2: Model Hashing Process

to resist model mutations (including the addition and deletion of model elements).

The principal requirement our fragment needs to meet is its statistical independence from the other fragments, so that a change affecting one fragment would not be propagated to other fragments, as this will reproduce the avalanche effect we precisely need to avoid.

Advanced model slicing techniques, aiming at extracting a subset from a model for a specific purpose (e.g., monitoring, model comprehension, modularity,...) [12] [13] [14] may be adapted to our use case. However, as we do not require our fragments to fulfill any *semantic* criteria, we propose a simpler approach, closer to approaches that automatically generate model fragments for the purposes of test generation [15] or efficient model storage [16].

[15] and [16] produce disjoint fragments. We need to drop this constraint so that model fragments are created in an independent way. In the same sense, we need to extract *many* fragments. In Definition 1, we show how we build fragments around a model element acting as *center*. The process of selecting a small number of centers would be greatly affected by mutations by imposing different orders to the list of existing models elements. We avoid this problem by extracting a large number of fragments (experiments show that the best results are obtained when the number of fragments approximates the number of model elements in the model) and by later using content-based classification to select just a few of these fragments for the final hash. We will see in Section 4.3 how the classification step guarantees that 1) we take fragments that are different, maximizing the coverage of the model to hash; 2) we take similar fragments even in the presence of mutations.

Definition 1. *Connected Model Fragment. Let A be the set of all model elements in a given model, $c \in A$ the model element center of the fragment and C^+ the transitive closure of neighbours of c (being neighbours any element reachable from c), then, we describe a fragment of A with center in c and size n:*

$$F_c^n(A) = c \cup \{x_i : x_i \in A, x_i \in C^+, 0<i<n-1\}$$

Remark. *Note that $|C^+|$ may be smaller or bigger than n. If smaller, all elements will be added. If bigger, n elements must be chosen using any suitable criteria.*

Top of Figure 3 shows a random fragment extracted from the ProMarte.ecore metamodel obtained from the ATL Metamodel Zoo[1]. The fragment has its *center* in the *TimingMechanism* class and includes four neighbours (*Clock*, *TimedEvent*, *MetricTimeValue* and *MetricTimeInterval*).

Algorithm 1: getNNeighbors()

```
Data: A; c; n;
Result: $F_c^{n-1}(A)$
$Neighbors \longleftarrow \emptyset$;
$Refs \longleftarrow c.getRelations()$;
for $ref \in Refs$ do
    $Neighbors \longleftarrow Neighbors \cup c.resolve(ref)$;
end
if $Neighbors.size() == n$ then
    return $Neighbors$;
else if $Neighbors.size() > n$ then
    return $\{x_i \in Neighbors, 0 < i < n\}$ ;
else
    $missing \longleftarrow (n - Neighbors.size())$;
    $Candidates \longleftarrow Neighbors$;
    while $missing$ != 0 and $Candidates.size()$ != 0 do
        $e \longleftarrow Candidates.getElement()$;
        $Candidates \longleftarrow Candidates - \{e\}$;
        $Neighbors \longleftarrow$
          $Neighbors \cup getNNeighbors(e, missing)$;
        $missing \longleftarrow (n - Neighbors.size())$;
    end
    return $\{x_i \in Neighbors, 0 < i < n\}$;
end
```

We present in Listing 1 a generic algorithm for the calculation of fragments. It takes as parameters: the model A; an object c, acting as fragment center; an integer n for the fragment size. Basically, it obtains all

[1] http://web.emn.fr/x-info/atlanmod/index.php?title=Zoos

the *relations* of the center element (line 2) and then resolves them (lines 3-5) in order to obtain a list of neighbors. If the size of this list is equal to n, we directly return the list. If it is bigger than n, n elements are selected and returned. Finally, if the size of the list is smaller than n, elements are selected from the list and, recursively, their neighbors are calculated and joined to the original list of neighbors until it has *size* $>= n$ or the exploration space is exhausted. Last, (at most) n elements are selected and returned. This algorithm gives priority to direct neighbors over those obtained through transitive navigation.

This generic algorithm is then refined for concrete models and/or modeling frameworks. This notably requires to provide implementations for the operations *getRelations()* and *resolve()* that basically define the concept of reachability, and to define a selection criteria for picking elements from the neighbors set. We show in Section 5 how it is done for EMF metamodels and EMF instances models (e.g., ATL model transformations).

4.2 Signatures of Model Fragments

Before hashing 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 then use the *minhash* technique to obtain the fragment signatures. We use content-based [17] identities of model elements as the base for the translation from model fragments to summary sets. We call these content-based identities *model summaries* and *model fragment summaries.*

Definition 2. *Model Element Summary Set. Let M be a model element and* $M.f_i()$ *with* $0 < i < n$ *a list of functions returning subsets of its contained data (this is, a subset of the image of M, considered as a vertex, under the function a defined in Section 3. e.g., the list of operations in UML models). We have*

$$SUMMARY(M) = \bigcup_{i=1}^{n} M.f_i()$$

As with the calculation of fragments, this general definition needs to be refined in concrete implementations in order to make the summary contain the relevant information for a given application domain. We discuss concrete implementation of summaries in Section 5.

As an example, a model element summary set for the *TimingMechanism* element in Figure 3, and supposing we only use the element name and its proper attributes, would be the set: {"TimingMechanism", "stability", "drift", "skew", "state"}.

From this point, the procedure of obtaining a robust hash from a given model is completely generic, and no further adaptations are needed for concrete application domains.

Definition 3. *Model Fragment Summary. Let* $F_c^n(A)$ *be a model fragment composed of n model elements* M_i*, we have*

$$SUMMARY(F_c^n(A)) = \bigcup_{i=1}^{n} SUMMARY(M_i)$$

Basically, the summary of a fragment is obtained as the unification of the summaries of the elements contained in the fragment.

Going back to our example in Figure 3, a fragment summary for our sample fragment is shown: **Fragment A Summary:** {"TimingMechanism", "TimedEvent", "state", "Clock", "MetricTimeValue", "stability", "MetricTimeInterval", "drift", "skew"} which effectively is the union of the model element summary sets of the model elements *MetricTimeInterval*, *Clock*, *TimingMechanism*, *MetricTimeValue* and *TimedEvent.*

Once we have our fragments represented as summary sets, our goal is to replace them with much smaller representations that we call *signatures.* In order for these signatures to be adequate for our robust hashing scheme, they must be robust themselves, this is: 1) equal elements have the same signature; 2) similar elements have similar signatures, so that we can obtain, combined with the next step, resilience against modifications; 3) very different elements obtain very different signatures. In order to solve this issue, we adapt here the *min-wise independent permutations locality sensitive hashing scheme (minhash)*, used as an efficient way to detect near-duplicate elements in large datasets.

The *Jaccard Index* (see Definition 4) is an indicator of the similarity between two sets.

Definition 4. *Jaccard Index. Let A and B be sets, the Jaccard similarity* $J(A, B)$ *is defined to be the ratio of the number of elements of their intersection and the number of elements of their union:*

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Remark. *J(A,B) is 1 when sets A and B are equal and 0 when A and B are disjoint.*

The virtue of the minhash is that we can compare minhash signatures of two sets and estimate the Jaccard similarity of the underlying sets from them alone. That is, the signatures preserve the *Jaccard Index* while being smaller and easier to manage than the original sets.

Basically, a signature for a given number of sets is built by: 1), creating an ordered dictionary of existing words for the universe of sets to hash; 2), creating a characteristics matrix by assigning 1 if the word exists in the set to hash and 0 if it does not exist. As an example, being the dictionary {this, is, a, dictionary} and two sentences to be hashed: ”a dictionary” and ”this dictionary”, the characteristics matrix would be

$$c = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Finally, the hashes are constructed by permuting (the rows of) this matrix and selecting, for each sentence to hash, the first row index with a non-zero value. The signature will have as many components as permutations. Having the following 4 permutations:

$$p1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} p2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} p3 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} p4 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}$$

We mark the first row index with a non-zero value in blue for the first sentence to hash and in red for the second one. The signature for the ’a dictionary’ will be [0,0,0,1] and for ’this dictionary’[2,1,0,0].

However, building a characteristics matrix (and corresponding permutations) for a real world application where the dictionary of possible words is very big is impractical. Therefore, the process is usually simulated with a number of random hash functions (as many as the elements the signature will have) in the following manner:

First, the minhash signature vector of size k (the number of hash functions to use) is initialized with *infinite* values. Then, for each word i in a set of words to minhash (a) the word i is transformed to a numerical representation (e.g., by using MD5); (b) for each hash function h_k, the numerical representation of i, $num(i)$ is hashed and $minhash[k]$ gets assigned the minimal value between $minhash[k]$ and $h_k(num(i))$.

We adopt here this latter approach. More formally, hash-based minhash signatures are defined as follows:

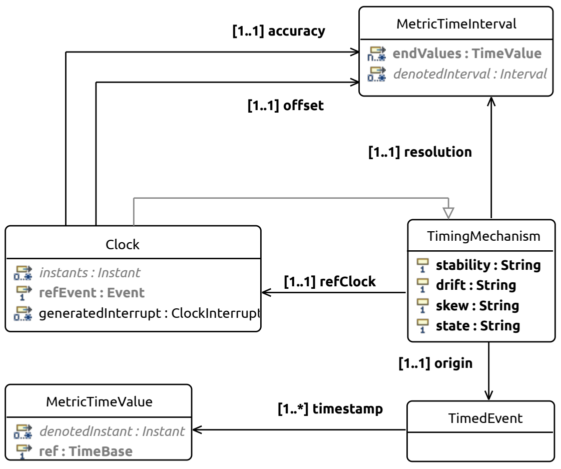
Definition 5. *Minhash Signature.*

– *Let h_i with $0 < i < k$ be a collection of hash functions and S a source set.*
– *Let $h_{min,i}(S)$ be the member x of S with the minimum value of $h_i(x)$*

Then, the signature of S is the vector composed of all the $h_{min,i}(S)$ with $0 < i < k$:

$$SIGNATURE(S) = [h_{min,1}(S), h_{min,2}(S), \ldots, h_{min,k}(S)]$$

Remark. *From [7] we know that for any i, the following equality holds. $Pr[h_{min,i}(A) = h_{min,i}(B)] = J(A, B)$. This is, the probability of $h_{min,i}(A)$ to be equal to $h_{min,i}(B)$ corresponds to the original* Jaccard index *between A and B. It is demonstrated then that the average $h_{min,i}$ taken as binary variables is an unbiased estimator for the* Jaccard index*. Thus, our signatures will preserve the* Jaccard index *of the original summary sets (with some bounded error, inversely proportional to k).*



Fragment A Summary:
{TimingMechanism, TimedEvent, state, Clock, MetricTimeValue, stability, MetricTimeInterval, drift, skew}
Fragment A Minhash Signature:
[3, 1, 2, 3, 0, 6, 1, 2, 4, 5, 1, 2, 0, 10, 0, 0, 5, 0, 2, 5]

Mutated Fragment A Summary:
{TimingMechanism, TimedEvent, state, MetricTimeValue, stability, MetricTimeInterval, drift, skew}
Mutated Fragment A Minhash Signature:
[3, 1, 2, 3, 0, 9, 1, 2, 4, 5, 1, 2, 0, 13, 0, 0, 5, 0, 2, 5]

Fragment B Summary:
{NFPCategory, QualitativeNFP, NFPLibrary, AnnotatedModelElement, Quantity, NFP}
Fragment B Minhash Signature:
[5, 8, 5, 1, 17, 4, 11, 12, 1, 2, 15, 3, 1, 3, 1, 4, 0, 2, 2, 4]

Fig. 3: Fragment Signatures

We show in Figure 3, below the fragment diagram, its corresponding *Model Fragment Summary* and *Minhash Signature* calculated using 20 hash functions. Then, we show the summary and signature of the fragment after a mutation that removes the *Clock* class. Finally, we show the summary and signature of a totally different fragment. From this example we can see that similar fragments (e.g., a fragment after a mutation) get very similar signatures (only positions six and fourteen of the signatures are different) while different fragments get totally different signatures.

4.3 Classification of Model Fragments and Hash Sequencing

The three previous steps have allowed us to transform a model into a large set of minhash signatures. As explained before, we have decided to generate a large number of these signatures to avoid model mutations influencing the hash creation process (e.g., modifying the order of selected elements). However, much of the information generated in this manner is very redundant, and thus, can not be efficiently used as is for the generation of the model hash. Instead, the idea is to, having a large list of classification buckets, take a certain number of different signatures by performing a classification step that puts similar signatures in the same bucket. Then we can just choose one of the elements of the bucket as its representative. Moreover, while model mutations may affect this selection step, the degree of the error would be minimized (if an element ends up in the same bucket, that means it is still quite similar) and not propagated across buckets.

This latter problem may be solved by the use of a locality-preserving hash function for our signatures. We need our signatures to form a *metric space*, this is, 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, e.g., the previously introduced Jaccard index). Then, if they exist, locality preserving hash function families will map elements belonging to such a metric space to a bucket $s \in S$.

Definition 6. *(d_1, d_2, p_1, p_2)-sensitive hash function. Let $d_1 < d_2$ be two distances in a given metric space, a family of hash functions 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 than d_1 is at least p_1; (ii) the probability of a collision when the distance is higher than d_2 is smaller than p_2 (this basically tells the probabilities of collision for very similar and very different elements).*

Our signatures are numeric vectors and as such, they form a metric space under the Hamming distance. In general, it is easy to obtain locality preserving hash functions for the vector metric space. As an 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 [18] that they form a $(d_1, d_2, 1 - d_1/d, 1 - d_2/d) - sensitive$ family. These simple functions are not very good for classification purposes as they only take one point of the vector into account and may produce a high rate of wrong collisions. However, the situation can be improved by building a new family of hash functions using a technique called *amplification* (the amplification technique builds new families of hash functions by combining existing ones with conjunction, disjunction or both). Note that this a very similar process as the one used for the construction of *MinHashes*.

As an example, imagine two vectors of size 10 have 9 positions with equal values. This makes the probability of classifying the vectors to the same bucket by using a single point hash function of 90% and the probability of a wrong collision when only one position is equal, of 10%. Conversely, by using a three point function the probabilities of hashing the vectors to the same bucket remain quite high (0,7%), but the probabilities of wrong collisions with low similarity drastically reduced.

Once the classification is done, the hash is constructed by polling the buckets to extract the desired number of fragment signatures.

4.4 Compression

The last step of our robust hashing approach is optional. If reducing the size of the final hash and/or providing it with extra robustness to model changes is needed, the use of a scalar quantizer is advised (see Definition 7).

A scalar quantizer assigns each value of the obtained hash to a given interval so that the number of bits required to represent them gets reduced. The process will also make the final hash more resistant to mutations as very similar hashes would have values that will fall in the same interval. Alternatively, although arguably more complex, error corrections codes may be used to obtain a similar effect by exploiting the fuzzy commitment technique [19] consisting in directly using the decode function without the prior encoding of the data. This way, the hash data would be treated as a message received through a noise channel, then *decoded* to assign it the *nearest* codeword in the given error correction code.

Definition 7. *k-level quantizer.*

- *Let d_i with $0 < i < k + 1$ be decision levels where d_i divides the range of data under quantization into k consecutive intervals $[d_0, d_1)[d_1, d_2)...[d_{k-1}, d_k)$.*
- *Let reconstruction levels $r_0, r_1, ...r_k - 1$ be the centers of the intervals.*
- *Let v be a value to quantize.*

– *Then,* $quantized(v) = i$ *when* $v > d_i$ *and* $v \leq d_{i+1}$

As an example of quantization, if we use twenty hash functions for the *minhash* signatures, we would obtain vectors of twenty elements with values from zero to nineteen, each of one requiring five bits for its representation. By creating intervals of just two consecutive values (e.g., we assign the values two and three to two) we will be able to represent each value with four bits, saving twenty bits for the whole minhash signature.

5 Implementation

We have described in Section 4 a generic method to create robust hashes for models. In this section we detail how this generic process is implemented, which parameters need to be set, and how to adapt the first two steps (namely, the extraction of fragment and the creation of fragment summaries) to concrete models and/or modeling frameworks.

Figure 4 summarizes the implementation architecture and the important parameters that guide the process. To sum up, a modeling framework- and/or model specific *Model Manager* reads the model in order to produce a list of fragments summaries. It takes as parameters: S, as the required number of fragments to extract and N, as their required size. Then, a *Minhash Calculator* takes the list of summaries transforming them to minhash signatures by using K hashing functions for the process (and thus producing signatures of size K). Finally, a *Hash Sequencer* reads R minhash signatures from a dedicated signature storage to produce the final robust hash for the input model. Note that, for simplicity, we do not implement the optional compression step. We devote the rest of this section to further describe the *Model Manager*, *Minhash Calculator* and *Hash Sequencer* components.

In order to ease the discussion, we give here the details of a concrete prototype implementation for the EMF framework. This prototype (available online[2]) includes the implementation of the model fragmentation, summary calculation, minhash signature generation and hash classification and sequencing. It has been implemented using Java and the Eclipse Modeling Framework (EMF) API [9].

5.1 Model Manager

The *Model Manager* is in charge of taking a model, extracting a number of *connected model fragments* (as presented in Definition 1) and transform them into summaries (see Definition 3).

5.1.1 Fragments

As mentioned in Section 4, the extraction of connected fragments is modeling framework and/or model specific. This is so because what is considered a model element and what is considered a connection is not generic. Nevertheless, Algorithm 1 is generic enough and only the input parameter A and two operations: *getRelations()* and *getResolve()* need to be implemented for concrete targets. In the following we explain how this is done for EMF models.

Figure 5 summarizes the main components (w.r.t. our use of it) of EMF. In the upper part of the figure we show an excerpt of the *Ecore* (meta) metamodel. Basically, it describes a language in which *EClasses*, contained in *EPackages*, have a number of *EStructuralFeatures* and *Operations*. *EStructuralFeatures* can be *EAttributes* or *EReferences*. With respect to the generic definition of models provided in Section 3, we consider *EClasses* as vertices and *EReferences* as edges, whereas *EAttributes* and *EOperations* are managed as attributes. The *eSuperTypes* relation is consider as an edge as well.

EMF allows us to define concrete metamodels conforming to the Ecore metamodel. As an example, in Figure 3 we show an fragment of the SysML metamodel, a general-purpose modeling language for systems engineering. Once the SysML metamodel is defined, we will be able to create instance SysML models, this is, a model consisting of concrete *Clocks*, *TimedEvents*, etc.

In the bottom part of the figure we show the main component of the runtime side of the EMF framework, which is the *EObject* object. Basically, all the elements in Figure 5(a) can be seen as *EObjects* when the model is loaded in order to be used and/or manipulated. This *EObject* provides, among others, four operations that allow us to navigate the model. Given a concrete *EObject*: *eContents()* gives a list of the elements related to the given *EObject* through a containment *EReference* (*eAllContents()* does the same but recursively in order to calculate the transitive closure of related elements); *eGet()* gives the value of an *EAttribute*; *eClass()* gives its metaobject (e.g., called on a SysML Clock instance of the metamodel shown in Figure 3, it would return the Clock *EClass*).

Thus, for EMF models, we consider *EObjects* as our model elements. Then, and considering *root* as the root element of a given model[3], the collection A is imple-

[2] https://gitlab.com/smartine/RobustModelHashing

[3] note that EMF models may have more than one root or just store the elements at the package level without root (we

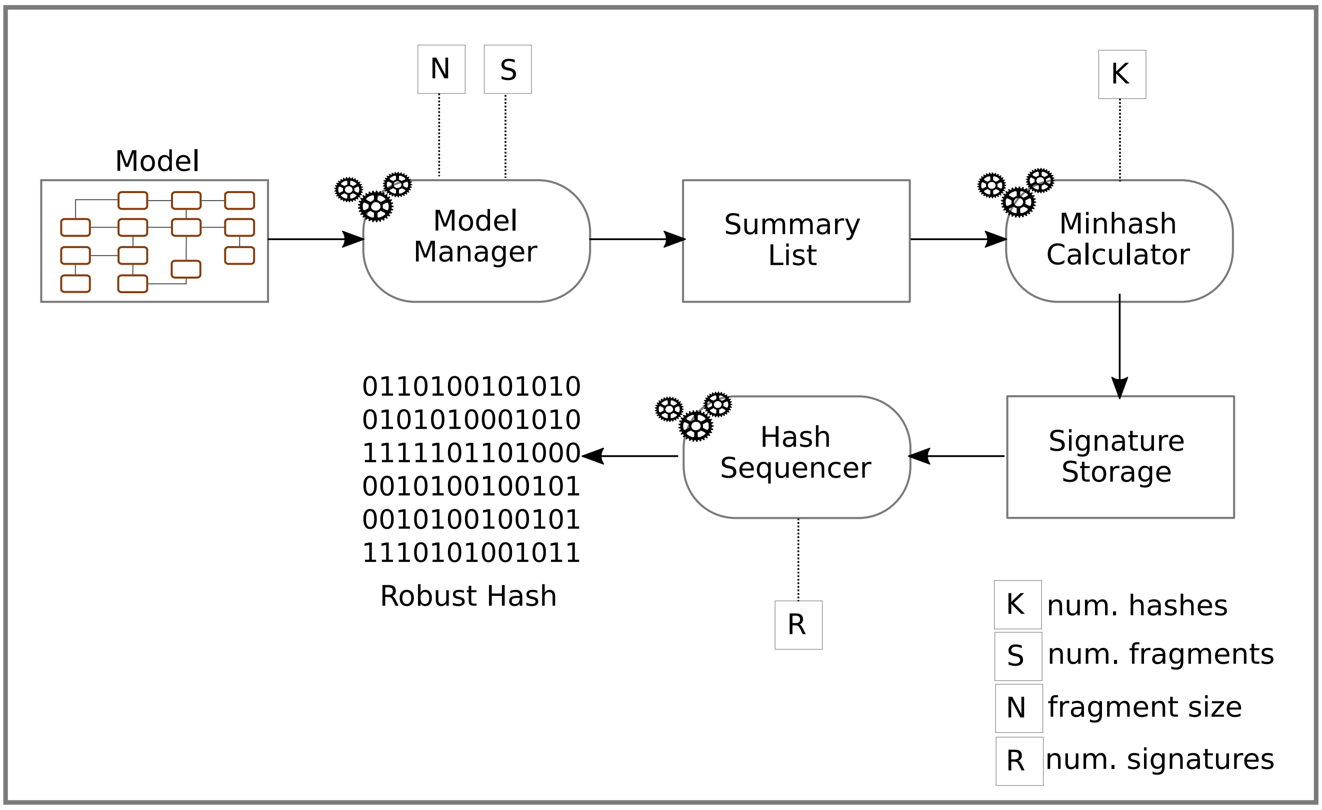


Fig. 4: Implementation

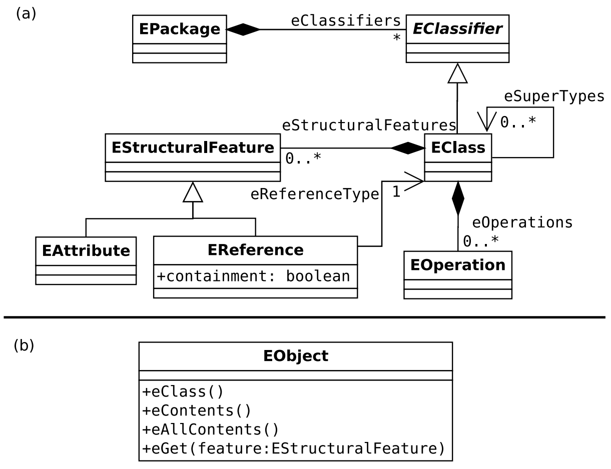


Fig. 5: EMF modeling framework excerpt

mented to contain the same elements obtained when traversing the iterator provided by the EObject method call *root.eAllContents()* (this is, all the EObjects of the model when applied to the root element). Note that in order to avoid ordering issues related to the modeling API traversal functions and/or model mutations, model elements should be stored on a content-based hash map so that the selection of fragments centers among executions is consistent (this is not necessary if we calculate many fragment centers, as the classification step performed for the sequencing is already content-based and thus, not affected by possible traversal ordering changes).

As for the *getRelations()* operation used to obtain the neighbors of a given *center*, we call via reflection the *center.eClass().getEAllReferences* to get the list of references the element may hold. Then, the *getResolve()* operation is implemented by calling *center.eGet()* on each reference. Note that we all also add to the neighbors set the element containing the *center* EObject.

Note that our approach is generic and that no restrictions are imposed beyond the existence of vertices and edges. As an example, ATL transformations [20], which consist in a number of transformation rules indicating how to transform source model elements in target model elements may be directly processed as EMF instance models by using (all) EObjects as vertices and (all) declared relationships as edges. However, alternative, more domain-specific, processing strategies may envisioned. In that sense Fragments may be constructed by using only rules and helpers as vertices, and the relations on a calculated dependency graph [21] as edges.

can then consider all those elements as roots). In that case, root contents are added sequentially to A.

5.1.2 Summaries of model fragments

Once we have decided which elements to take into account for the robust hashing we need to translate them to sets of *words*, these words representing its identity and contents. This translation is domain specific as different summaries can be created from the same elements depending of what feature is to be highlighted.

As an example, for EMF model elements, i.e., for each *EObject*, we create the summary set by inserting:

– The model element's *type* name.
– All direct (e.g., non inherited) element's attribute values (transformed to strings)

Similarly, for Ecore metamodels we create for each *EClass* the summary set by inserting:

– The model element's name.
– The model element's list of super types (i.e. the list of elements it is declared to extend).
– All direct element's declared attribute names.
– All direct element's declared operation names.

These are very generic summaries, aimed to work with any model belonging to the EMF modeling framework. Note however that more specific summaries may be constructed by adding or filtering different information (e.g., filtering recurrent attributes such as models IDs, etc). The importance of the adaptability for comparison tools has been discussed in [22].

Once we have the model element summaries we can create the fragment summaries as a simple union of the summaries of the fragment's members. Note that relations information is not directly included in the model summaries. Relations between elements are implicitly captured by the concept of connected fragment. This is, all elements in a fragment are connected with one or another kind of relation. Relations may be added explicitly to the fragments (or the individual model fragments). Note however that adding many details of the relation, such as cardinalities, etc., will increase the discrimination capacity of the approach but decrease its robustness. In the same sense, information about the ownership of attributes is lost when using very generic summaries, but may be easily added if needed.

5.2 Minhash Calculator

The implementation of the minhash signatures is straightforward. Each letter in a string is transformed into a numerical value and hashed. Hashes for a given string are accumulated by using a *XOR* operation. For the generation of the required hash function for the simulation of the permutations we take random k hash functions from the universal family $ax + b\%p$ (with p being a prime number bigger than the desired number of permutations k and a, b integers smaller than p).

5.3 Hash Sequencer

Practically, the classification step is performed by using an and-augmented locality preserving hash function (see Definition 6) together with the java *hashcode()* operation for vectors and a modulo operation on the number of buckets. This makes similar fragment signatures collide in the same bucket and different signatures distribute along the storage hash map (given a sufficiently large hash map).

Once the classification is done, the hash is constructed by polling the buckets randomly with the help of the secret key in order to take the desired number of minhash signatures. Note that in case of selecting an empty bucket, the polling algorithm will just take a minhash signature from the next non-empty bucket (the same list of buckets will be selected across execution, assuring us to always take similar minhashes when we are dealing with similar models).

6 Experiments & Evaluation

We devote this section to the analysis and experimental evaluation of our model hashing approach. The robust hashing properties presented in Section 2: *discrimination*, and *robustness*, are analyzed to assess how well our approach fares against them.

We start by evaluating the influence of the parameters described in Section 5 in the accuracy on the similarity estimation. Then, for each of the aforementioned properties we discuss the expected behavior as derived from the construction of our approach and the actual validation using our prototype implementation.

6.1 Parameter Evaluation

As discussed in Sections 4 and 5, our robust hashing approach for models depends on four main parameters: the number of hash functions used for the calculation of the minhash (K); the number of fragments to extract from a given model (S); the size of the extracted fragments (N); and the number of signatures taken in order to build the final robust hash (R). We are interested in finding boundaries for these parameters, this is, minimum usable values, and max values from where extra investments do not improve outputs.

6.1.1 Determining the number K of hash functions for the minhash calculation

In order to choose K, we have to take into account that the signatures estimate the Jaccard index for sets. With larger values of K, we get better discrimination between sets and less *collisions* for not sufficiently equal fragment summaries. On the other hand, a larger K generates longer signatures what will lead to bigger hashes.

In order to illustrate the effects of choosing different values for K, we have evaluated how well the similarity between model fragments is approximated with the different values. First, we have selected a random model from the ATL metamodel zoo (concretely, *OpenConf.owl.ecore*) extracted 20 fragments of size 3 from it, and generate the corresponding summaries. Then, we have done a pairwise comparison among all the summaries by using the Jaccard index (see Definition 4). The results of this pairwise comparison are shown at the top of Figure 6 (note that we have scaled the Jaccard index to 100). It can be seen that a few pairs of summaries have a high similarity while most of the pairs have low or even zero similarity. Ideally, we should obtain very similar results when comparing minhash signatures obtained from the aforementioned summaries. We have tested this hypothesis by: 1) generating sets of minhash signatures for the corresponding summaries with different values of K (concretely $K = 200, 100, 50, 20, 10, 5$); 2) making a pairwise comparison among all the signatures pairs in a given set by using the Hamming similarity measure (see Definition 8). Figure 6 summarizes the obtained results. Note that by using $K = 200$, the similarity estimation is very accurate. As we lower the value of K, the accuracy degrades.

Definition 8. *Hamming Similarity. Let s and t be numeric vectors, the Hamming similarity is:*

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

A priori, the bigger the K, the better. Note however that there are two factors to be taken into account: first, lower K values specially fail at estimating when summary pairs have a low similarity; second, we are mostly interested in a good separation between highly similar pairs and the rest (e.g., the concrete similarity value is not important as we will never be making comparisons with the real similarity). In that sense, and for the model at hand, we need to discard using $K < 10$ as the separation between high level similarity pairs and the rest can not be done. However for $K = 20$ we already obtain a good separation.

We have seen how bigger $K's$ estimate better the similarity between fragments of size 3. Intuitively, we can guess that working with bigger fragments would need bigger values of K. We have investigated how the size of the fragments affects similarity. In order to do so we calculated the average difference between the Jaccard index and the Hamming similarity for each K. We show the results of this calculation in Figure 7. As it can be seen, the size of the fragments affect the accuracy of the similarity estimation, being the effect stronger for lower values of K. In that sense, low K values are to be avoided when dealing with big fragments.

6.1.2 Determining R, the number of signatures to take for the final hash

Unlike K, that is used at the fragment summary level, R, the number of signatures to take in order to form the final hash needs to be tested at the model level. This means that we have to compare the similarity calculated on the model to the similarity calculated directly on the robust hashes. Unfortunately, while a standard metric for the robust hashes exists (Hamming distance for vectors), this is not the case for models in general. Domain specific metrics [23] and diverse shape-based graph distance metrics [24] do exist. However, they are either not applicable to every model or just do not take both, content and shape into account at the same time. Generic approaches for models taking into account shape and content have been proposed for models [25, 26], but they do not focus on yielding a similarity or dissimilarity measure value, but in matching *similar* elements within models and/or building a differences structure that records what changes from one model to another, so that it can be operated afterwards (probably because many of those approaches were proposed for the use on versioning control). More similar to what we need are the distance measures defined in [10], but they focus on the particular case of models where some subset of model elements *moves* around another subset of fixed model elements. Moreover they require the semantics of the *movement* encoded as in-place model transformation rules. Finally, some similarity measures (such as Hamming, cosine and Levenshtein) are defined in [27, 28]. However, they work on a vector representation of the models and not the models themselves (besides, whether their vectorization of models preserves proximity is not discussed).

In order to tackle this problem we have decided to rely on a widespread tool for the comparison of models, EMFCompare [29]. EMFCompare does not provide a similarity value when comparing two models. Instead, it provides a list of differences (which can be seen as an adaptation of the Levenshtein distance to the model differencing domain). The more different two models are,

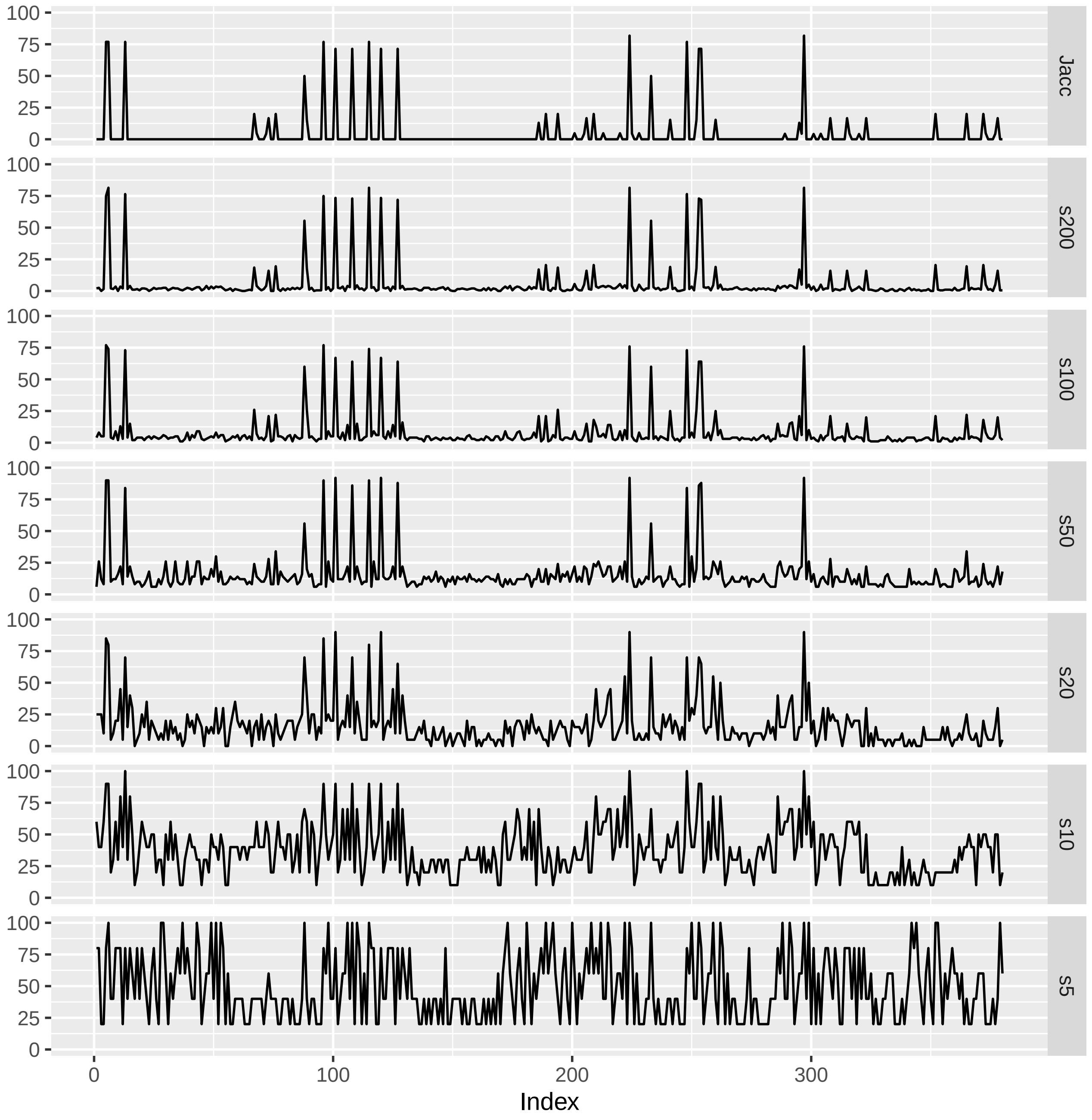


Fig. 6: K Evaluation

the longer is the list of differences. The general hypothesis is that the value of robust hash similarities decrease as the list of EMFCompare differences increase when a pairwise comparison between models is performed. This means that: 1) a negative correlation exist between both metrics; 2) the strength of the correlation depends on R.

In order to evaluate this hypothesis, we have created a handful of instances of PetriNet models (we use a variation of the Petri Net metamodel described in [30] so that models can contain many different Petri Net and associations between Petri Nets are allowed) of a uniform size of about 150 model elements.[4] Then, we have randomly introduced mutations to these models in order to create repositories of models with a diverse degree of similarity. The introduced mutations include: adding and removing objects and modifying attributes

[4] the use of the EMFCompare differences list's size as metric, requires the use of models of similar size for it to be meaningful.

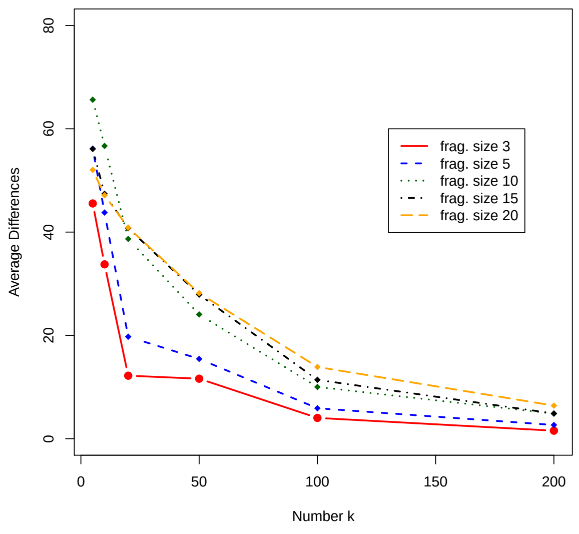


Fig. 7: (Negative) Average difference from Jaccard Index depending on K

and reference values (we use an adaptation of the Ecore-Mutator tool[5]).

Finally, for each pair of models in the repositories we calculated their lists of differences with EMFCompare and the similarity of their corresponding robust hashes by using the Hamming similarity measure (see Definition 8).

Figure 8 summarizes the results of this evaluation (the x-axis shows EMFCompare differences whereas the y-axis shows robust hash similarity). As it can be seen, there is a strong negative correlation between both metrics [6], and this even by using a low number of signatures to form the hash. Using more signatures improves the strength of the correlation but this improvement comes at the price of having to store and manage larger hashes. In the general scenario, where distinguishing very similar from very dissimilar models is the objective, building small hashes with a relatively low number of signatures (20 to 50) performs well enough.

6.1.3 Determining N, the fragment size

We have seen that using larger fragment sizes requires using a larger number K of hash functions for the calculation of the minhash but that lower values present already a good separation between very similar and dissimilar model fragments. We are interested in evaluating whether this result generalizes to the whole robust hash once built by using a certain number of fragment signatures. Concretely, and given a value for K and R, we are interested in evaluating whether the size of the fragments influences the strength of the correlation between EMFCompare differences and robust hashes' Hamming similarity.

In order to evaluate this hypothesis, we have repeated the setting of the previous evaluation (e.g., pairwise comparison at the model and robust hash level of a set of repositories of mutated Petri Net instance models with different degrees of similarity between them) with $K = 30$, $R = 50$ and N in the set $\{5, 10, 15, 30\}$.

Figure 9 summarizes the results of this evaluation. As it can be seen, the correlation between both metrics remains strong for all values of N, but it weakens with $N > 10$ which confirms that the effect of choosing a given size of K generalizes to the whole robust hash (i.e., as fragment summary minhashes lose accuracy, the whole robust hash does it as well). Summarizing, to maximize accuracy, N should be chosen to be as big as possible but only if K is sufficiently large as well (see Figure 7).

6.1.4 Determining S, the number of fragments to create

As for the number S of fragments to create, we need just enough fragments so that we can take R different signatures to build the final hash. Generating more fragments would not directly affect the quality of the final robust hash.

Note however that taking S = (desired R) does not guarantee to have R fragments, as the fragment content-based classification step will put very similar fragments in the same bucket, and we only take one fragment from a given bucket. Thus, S needs to be larger than R. A practical solution would be to generate the robust hash with a given $S > R$, verify its size, and regenerate with a bigger S if the robust hash is smaller than R.

We have evaluated how the configurable parameters of our approach affect its similarity estimation accuracy w.r.t. similarity metrics calculated on the fragment summaries and whole models (i.e., Jaccard index and EMFCompare differences). In that sense we believe that a good set of parameters is $K = 30$, $S = 200$, $R = 100$ and $N = 10$ as: 1) we already see good separation from $K > 20$; 2) We do not see an improvement when using fragments bigger than $N = 10$ (besides, the whole process relies of the idea of having many different fragments instead of a few overlapping ones); 3) We obtain the best results with $R = 100$, but not so much better

[5] https://code.google.com/archive/a/eclipselabs.org/p/ecore-mutator

[6] we used Spearman rank correlation obtaining a coefficients ranging from 81 to 90.

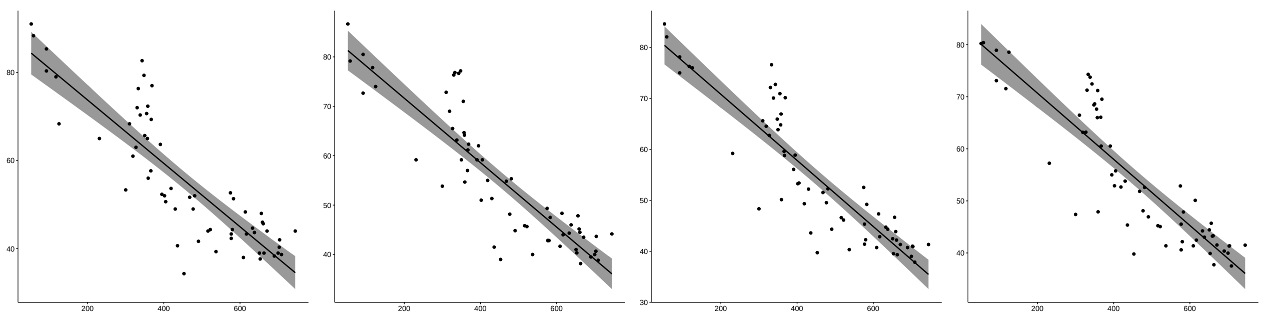

(a) 10 Signatures (b) 20 Signatures (c) 50 Signatures (d) 100 Signatures

Fig. 8: Evolution of correlation w.r.t. number of Signatures

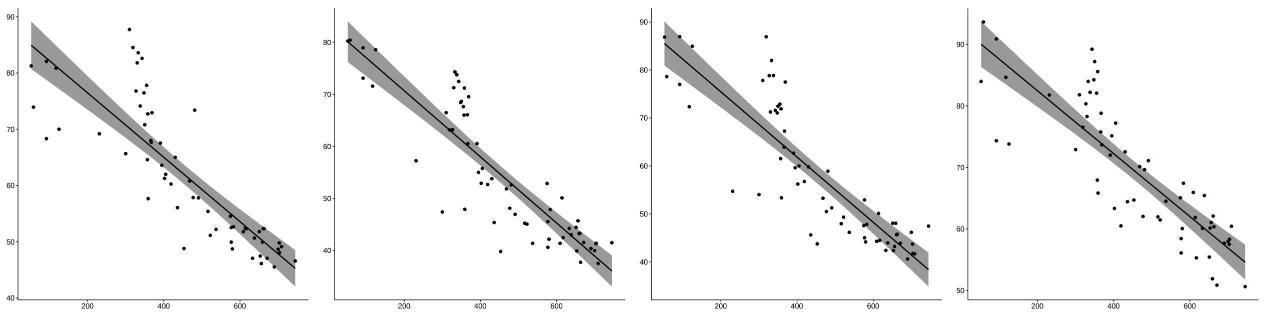

(a) 5 elements (b) 10 elements (c) 15 elements (d) 30 elementes

Fig. 9: Evolution of correlation w.r.t. the fragment size

w.r.t. to $R = 50$ to make using even more signatures worthy; 4) finally, we set $S = 200$ so that having at least 100 different signatures to take is highly probable.

Below we will discuss how these general results relate to the properties of *discrimination* and *robustness*.

6.2 Discrimination

Discrimination is the ability of a robust hashing algorithm to produce different hashes for (very) different models. Effectively, while we want our robust hashing algorithm to resist modifications, it would be useless if it can not tell when two models are not related. We achieve a high level of discrimination by using connected model fragments as the basis for our feature extraction. Indeed, by working with fragments and not with isolated model elements, our approach is able to distinguish between models using similar vocabulary but with a different structure.

We have seen in the evaluation of the parameters that our approach does discriminate between very similar and dissimilar models (see Figures 8 and 9) for models of the same kind. In the following we intend to evaluate if the discrimination property holds for models that are completely different. In order to do so, we have: 1) selected 18 different metamodels from the ATL metamodel zoo; 2) hashed them with our robust hashing algorithm (We use $K = 30$, $S = 200$, $R = 100$ and $N = 10$ by following the parameter evaluation results of section 6.1); and 3) calculated the similarity of the hashes between all pairs.

We show in Table 1 the result of the pairwise similarity calculation (multiplied by 100). The darker the cell, the more similar the hashes. As we can see, only the hashes obtained from the same model get higher levels of similarity, while the hashes of different models get similarities normally lower than 20. It is interesting to see that the higher similarity value between two different models corresponds to the pair MICRO.owl.ecore - OpenConf.owl.ecore that reach a similarity coefficient of 62. In fact, MICRO.owl.ecore and OpenConf.owl.ecore are indeed two different versions of the same metamodel (the second one being an extension of the first one).

Summarizing, our robust hashing algorithm did not lead to false positives and was able to detect the only pair of derived models. Therefore, we can conclude that the discrimination property holds for our approach.

Table 1: Pairwise Similarity

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
a	100	7	6	9	7	9	9	8	8	7	7	8	9	7	8	8	8	7
b	7	100	10	9	7	11	11	10	8	9	8	9	12	8	10	7	10	10
c	6	10	100	9	8	9	12	10	9	9	8	8	12	10	11	8	10	9
d	9	9	9	100	8	11	12	10	9	10	9	8	14	9	12	9	12	10
e	7	7	8	8	100	10	9	9	9	7	8	7	10	9	9	7	9	8
f	9	11	9	11	10	100	15	15	10	10	11	10	15	11	11	9	10	11
g	9	11	12	12	9	15	100	13	12	12	11	10	17	11	14	11	13	11
h	8	10	10	10	9	15	13	100	9	11	9	11	14	9	11	7	11	10
i	8	8	9	9	9	10	12	9	100	9	8	9	12	9	10	8	9	8
j	7	9	9	10	7	10	12	11	9	100	9	8	13	9	10	8	9	10
k	7	8	8	9	8	11	11	9	8	9	100	7	11	9	9	8	9	9
l	8	9	8	8	7	10	10	11	9	8	7	100	9	8	8	8	9	9
m	9	12	12	14	10	15	17	14	12	13	11	9	100	10	16	9	14	13
n	7	8	10	9	9	11	11	9	9	9	9	8	10	100	10	8	9	10
o	8	10	11	12	9	11	14	11	10	10	9	8	16	10	100	7	62	10
p	8	7	8	9	7	9	11	7	8	8	8	8	9	8	7	100	8	8
q	8	10	10	12	9	10	13	11	9	9	9	9	14	9	62	8	100	11
r	7	10	9	10	8	11	11	10	8	10	9	9	13	10	10	8	11	100

*a:SBVRvoc; b:mlhim2; c:Agate; d:sbvrEclipse; e:J2SE5; f:Matlab; g:UML2; h:SCADE; i:XHTML; j:KDM; k:Maude; l:MoDAF-AV; m:ifc2x3; n:MavenMaven; o:OpenConf.owl; p:ProMarte; q:MICRO.owl; r:SWRC;

6.3 Robustness

Once we have shown that our robust hashing approach is not prone to false positives we proceed here to discuss its robustness, this is, its ability to resist modifications. As discussed in Section 2 our approach should resist to two types of modifications: 1) modifications related to the storage mechanism; and 2) modifications of the model content.

Our approach is robust against changes in the storage of the model as the storage information is not taken into account in the hashing process. As long as the implementation can provide model fragments as sets of model elements, our approach leads to the same results. In the same way, our approach is independent of the modeling framework used for the specification of a model (e.g., UML models, EMF models, GME [31],...). Note however that if required our approach could be sensitive to the specificities of a given modeling framework by including those specificities in the fragment summaries.

As for the modifications to the models contents and structure, our approach provides two protection mechanisms. First, the very use of *minhash* for the calculation of model fragment signatures gives robustness to our approach as similar fragments summaries will get the same or a very similar minhash. This is illustrated in Figure 3 where a fragment and its mutated counterpart obtained very similar signatures (this effect can be tuned, concretely, by choosing lower numbers K we can reduce the "resolution" of minhashes as seen in Figure 6.)

Table 2: Mutation Resistance

	Number of Mutations			
Model Name	5 mt.	10 mt.	25 mt.	50 mt.
ProMarte	91	82	77	52
Uml2	95	89	70	61
Scade	99	89	77	65
OpenConf	97	82	87	84
Matlab	85	69	48	40

Second, the use of locality-preserving hash functions for the classification of signatures w.r.t. their content before its use for the formation of the final robust hash provides robustness to our approach as it promotes the selection of the same (or very similar) model fragment signatures for the composition of the final model hash even in the presence of mutations.

To conduct the experimental evaluation of the robustness property we perform the hashing of five different regular-sized meta-models (we use the same configuration parameter's values as in the discrimination evaluation). Then, we compare the hashes of the original model to those of mutated versions to determine whether their similarity level allow us to conclude that the models are derivations.

Table 2 summarizes the obtained results (which are consistent with the results showed in Figures 8 and 9 for the correlation between the robust hash similarity and a EMFCompare-based metric).

Rows indicate the model, while columns indicate the number of introduced mutations (we introduce 5, 10, 25 and 50 mutations). Note that mutations are introduced randomly and independently between rows, i.e., we start always from the original model. This explains why for OpenConf.owl.ecore, a mutated model with 10 mutations get lower levels of similarity that a mutated model with 25 mutations. From the results we can see that models resist very well the introduction of up to 25 mutations, with similarity values much higher than the ones obtained for different models as shown in Table 1. The last column shows how after 50 mutations models become too different to be considered similar. Notably Matlab.ecore, that is the smaller model (with around 34 model elements each), gets the biggest impact.

As a summary, we can conclude that our robust hashing algorithm resists mutations well. Models need to be mutated to the extent of making them unusable for their original purpose in order to obtain very different hashes. Therefore our hashing algorithm is, as intended, robust.

6.4 Additional experiments

We have shown in Sections 6.2 and 6.3 that discrimination and robustness properties hold for Ecore models (extracted from the ATL Metamodel Zoo) when using the parameters obtained in Section 6. In order to show that the aforementioned properties hold with the same parametrization for a wider range of scenarios, here we extend the experimentation to different scenarios and different types of models. More specifically, we have created three different repositories containing each a different type of model. All repositories are composed of ten different models plus two models that are mutated versions of a model randomly selected from the former set of ten elements. The discrimination and robustness properties will hold if within each repository the different models appear as different and the mutated models are recognized as derivations.

The first repository contains ten different (generated by using the same procedure and metamodel as described in Section 6.1.2) PetriNet models together with two mutated versions of the ninth model (09_petri_05 and 09_petri_30). The second repository contains ten different Ecore metamodels randomly extracted from the MAR repository [32], which includes around 17.000 Ecore metamodels crawled from Github and the AtlanMod Zoo. Note that we impose three condition for the selection of a metamodel: 1) it must contain elements and not only package declarations (so that the robust hashing process can be launched); 2) it should not be present in the AtlanMod Zoo; and 3) it must be different from the other selected metamodels. As with the previously described PetriNet repository, one of these ten models is selected randomly to be the source of two mutated versions. In this case, it is the fourth metamodel (JKind.ecore). Finally, the third repository contains 10 model transformation models extracted from [33] together with two mutations of the first model transformation (tr01_m1 and tr01_m2). Note that, for each repository, the two mutated version include one slightly mutated (the first one in each case) and one more heavily mutated version.

Tables 3, 4 and 5 show the results of calculating the pairwise similarity (using our approach with the parameters $K = 30$, $S = 200$, $R = 100$ $N = 10$) for PetriNet, Metamodels and Model transformation respectively. As it can be seen, mutated models are easily spotted from the rest of the models in all the cases, showing the properties of robustness and discrimination hold.

6.5 Performance Evaluation

While the capacity of robust hashing approaches to be metric preserving is essential, the efficiency of the robust hash generation process is critical as well. Indeed, typical usages of robust hashing techniques may include: 1) the hashing of entire model repositories in order to find similar elements; 2) the hashing of large models in the aforementioned repositories. In these scenarios, a slow robust hash method will defeat the purpose of using a similarity estimation technique. Therefore, and in order to validate the suitability of our approach for the aforementioned usages, we present in the following a set of performance evaluation experiments. Concretely, we evaluate how our robust hashing method scales when parameters (representing work to be done) and model sizes grow[7].

The first step of our performance evaluation is to obtain a sample repository with a large range of model sizes, including relatively small and very large models. We do so by using the Atlanmod Model Instantiator [8]. We have used this instantiator to produce instances of PetriNet models of different sizes, from 100 to 1 million elements.

Then, we have taken a fairly large model (e.g., containing about 500K elements) and evaluated how the time for calculating its robust hash evolves as two parameters, the number of calculated fragments and their

[7] Experiments are performed on A Intel® Core™ i5-6200U CPU @ 2.30GHz × 4 cores, running Ubuntu 16.04

[8] `https://github.com/atlanmod/mondo-atlzoo-benchmark/tree/master/fr.inria.atlanmod.instantiator`

Table 3: Pairwise Similarity for Petri Nets

	a	b	c	d	e	f	g	h	i	j	k	l
a	100	40	42	41	39	41	40	40	43	40	40	38
b	40	100	42	43	45	40	40	40	41	43	41	39
c	42	42	100	44	42	45	45	44	43	42	40	39
d	41	43	44	100	42	44	42	42	43	42	41	39
e	39	45	42	42	100	44	41	45	42	42	41	42
f	41	40	45	44	44	100	44	44	42	41	42	40
g	40	40	45	42	41	44	100	43	44	44	42	39
h	40	40	44	42	45	44	43	100	44	44	42	40
i	43	41	43	43	42	42	44	44	100	71	69	41
j	40	43	42	42	42	41	44	44	71	100	66	40
k	40	41	40	41	41	42	42	42	69	66	100	40
l	38	39	39	39	42	40	39	40	41	40	40	100

*a:01_petri.xmi; b:02_petri.xmi; c:03_petri.xmi; d:04_petri.xmi; e:05_petri.xmi; f:06_petri.xmi; g:07_petri.xmi; h:08_petri.xmi; i:09_petri_00.xmi; j:09_petri_05.xmi; k:09_petri_30.xmi; l:09_petri_800.xmi;

Table 4: Pairwise Similarity for Metamodels

	a	b	c	d	e	f	g	h	i	j	k	l
a	100	11	14	12	8	16	9	10	7	12	11	11
b	11	100	12	10	10	12	12	9	5	16	9	10
c	14	12	100	11	9	13	11	10	6	15	11	10
d	12	10	11	100	9	11	9	9	7	11	82	62
e	8	10	9	9	100	13	10	8	6	13	8	9
f	16	12	13	11	13	100	11	10	7	16	10	10
g	9	12	11	9	10	11	100	9	6	13	8	10
h	10	9	10	9	8	10	9	100	7	11	8	8
i	7	5	6	7	6	7	6	7	100	6	7	7
j	12	16	15	11	13	16	13	11	6	100	10	11
k	11	9	11	82	8	10	8	8	7	10	100	66
l	11	10	10	62	9	10	10	8	7	11	66	100

*a:architecture.ecore; b:BusinessDomainDsl.ecore; c: CASL.ecore; d:JKind.ecore; e:svg.ecore; f:jobSearch.ecore; g: Language.ecore; h:Rell.ecore; i:restrain.ecore; j:xtce.ecore; k:JKind.ecore; l:JKind.ecore;

size (we evaluate these two parameters as the other two, namely the number of hashes for the minhash and the final number of used signatures, should not grow much), independently grow.

Figure 10 shows the results of this evaluation. Note that each tick mark in the x-axis supposes an increment of 1000 to the parameter and that the initial parameter values are: fragment size $N = 10$ and number of fragments $S = 100$. As it can be seen, extracting large number of fragments from a model scales very well, with the calculation of 10000 fragments taking less than 2 seconds. This is less so for the extraction of increasingly bigger fragments, with the generation of 100 fragments containing 10000 elements taking almost 20 seconds. This is explained by the nature of Algorithm 1 described in Section 4. Indeed, the *getNNeighbors* algorithm will try to recursively find 10000 neighbors of a given element, which, given that normally model elements do not have many direct neighbors, requires many recursive calls, model traversals and resolution. Nevertheless, the extraction of such big fragments over such big models is a corner case and many different strategies may be used in order to optimize the aforementioned algorithm (e.g., the incremental storage of calculated model neighbours in a map structure).

Next, we are interested in evaluating whether the size of models influences the time for calculating the robust hashes, all the other parameters being fixed. Figure 11 summarizes the obtained results. As it can be seen, model size do not have an important impact in the hashing times other than increasing the model loading time.

From this results, we can conclude that our approach scales very well and can be applied to very large models. The bottleneck is in the neighbors calculation algorithm used for the fragment extraction, that nevertheless performs very well for fragment sizes of up to 4000 model elements. Further performance evaluation, e.g., the gain of performance of using robust hashes instead of standard compare tools for determining general

Table 5: Pairwise Similarity for Model Transformations

	a	b	c	d	e	f	g	h	i	j	k	l
a	100	95	54	45	40	45	44	43	45	44	41	47
b	95	100	54	45	40	45	45	43	45	43	41	47
c	54	54	100	47	40	44	46	43	44	44	43	47
d	45	45	47	100	42	44	44	42	44	43	42	45
e	40	40	40	42	100	43	43	41	43	42	41	42
f	45	45	44	44	43	100	46	42	47	45	44	45
g	44	45	46	44	43	46	100	42	47	46	46	46
h	43	43	43	42	41	42	42	100	43	42	41	43
i	45	45	44	44	43	47	47	43	100	48	46	46
j	44	43	44	43	42	45	46	42	48	100	44	47
k	41	41	43	42	41	44	46	41	46	44	100	42
l	47	47	47	45	42	45	46	43	46	47	42	100

*a:tr01_m0.xmi; b:tr01_m1.xmi; c:tr01_m2.xmi; d:tr02.xmi; e:tr03.xmi; f:tr04.xmi; g:tr05.xmi; h:tr06.xmi; i:tr07.xmi; j:tr08.xmi; k:tr09.xmi; l:tr10.xmi;

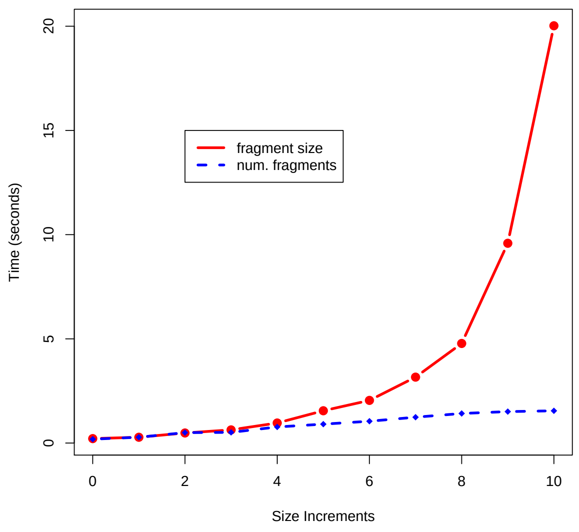


Fig. 10: Performance vs fragment size and number of fragments

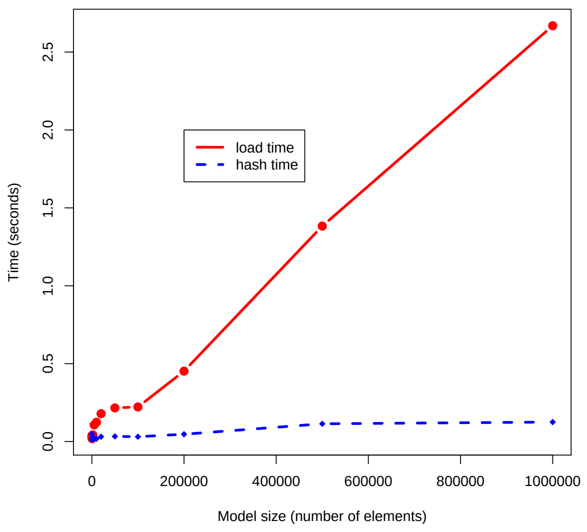


Fig. 11: Performance vs model size

similarity between pairs of models is out of the scope of this paper. Nevertheless such a performance evaluation is provided in [33] where it is shown that using the robust hashes for the comparison is orders of magnitudes faster.

6.6 Threats to Validity

The validity of the conclusions obtained by our experiments may be affected by:

(1) The absence of standard model similarity metrics: effectively, we evaluate our approach by comparing the similarity of models as estimated from our robust hashes with one metric based on the size of model differences which is not a standard metric and only valid for comparing models of similar size. We alleviate this threat by extending our evaluation to a large set of different models of varied size and by evaluating the accuracy at the fragment summary level, where a standard metric does exist.

(2) The mutation generation: different kinds of mutations are randomly introduced in models. It is difficult to ensure that a model with more mutations has been subjected to more modifications than another with less but more severe mutations (e.g., a class deletion mutation affects a model more than an attribute modifi-

cation mutation). Nevertheless, from the experimental evaluation we can conclude that the variations due to the random process are less important when the number of introduced mutations grows.

(3) The existence of a wide diversity of model types: from structural to behavior models passing by models representing software artefacts such as programming language code or configurations scripts, the characteristics of models vary greatly (in size, connectivity, information contained its elements, etc.).

We mitigate this threat by extending our evaluation to three different types of models, namely: EMF Ecore metamodels, ATL model transformations and PetriNet instance models, and showing the conclusions hold for the three of them. Section 6.4 shows this evaluation.

6.7 Usage & Adaptation

Section 6.1 was devoted to the evaluation of the influence of the parameters that guide our approach (see Section 5) w.r.t. the accuracy of its similarity estimation. From that evaluation a configuration was determined: $K = 30$, $S = 200$, $R = 100$ $N = 10$. Sections 6.2 and 6.3 show that these configuration values achieve good *discrimination* and *robustness* results for EMF Ecore metamodels of diverse size.

These values are meant to work with a wide range of models (models of different sizes and types). In that sense, we showed in Section 6.4 a number of additional experiments. Tables 3, 4 and 5 show discrimination and robustness (we have introduced mutants in each repository, which our approach detected as so by calculating a high similarity value for them) results from three different types of models: ATL model transformation models, PetriNet Models and GitHub mined metamodels. As it can be seen, a good separation is obtained between very similar models and the others which is the main objective of our approach.

Nevertheless, we may be interested in augmenting the distance between similar and dissimilar models in order to obtain a better *resolution*. We discuss below how to achieve that.

Improving resolution.

There are two main reasons for a lower than expected separation between similar and dissimilar models: The model size and the model characteristics. Indeed, very large models may need larger hashes in order to improve coverage while some particular types of models may need adaptation. In that sense [22] discusses the interest of adaptation of model comparison tools, so that they take into account particularities such as the reduced number of properties of some types of elements such as control nodes in activity diagram models or pins and ports in component diagrams.

Table 6 summarizes the mechanisms of adaptation in case on insufficient separation depending on the model characteristics. If the model is very large, the number of fragments to create (S) and to use for the final hash (R) should be increased. If size is not the issue, then, it is the model particularities that play a role in the insufficient separation. In that sense, while still sufficient, we can see that models in Tables 3 and 5 present a lower separation than the models in Table 4. Indeed, ATL and Petri Net models are models where structures repeat often and with model elements that do not contain too much information (e.g., few attributes, few relations). The adaptation required to improve separation in these cases are listed, in order of preference, in Table 6:

1. reduce the size of fragments (N). Indeed, by reducing the size of fragments, the hash of the signature will estimate similarity more accurately for a fixed K;
2. with the same rationale as previous one, augment the number of hash function for the minhash (K). We recommend first to try reducing N, as augmenting K results in larger hashes;
3. adapt the summaries so that they include different information (e.g., removing non relevant attributes).
4. adapt the fragment calculation so that more meaningful fragments for the specific case are generated.

Adaptations 1 and 2 are easier to apply and thus, should be tried first.

Evaluation of adaptations.

Tables 7, 8 and 9 show the result of applying adaptations 1, 2, and 3 to the transformation models of Table 5. Concretely, Table 7 shows the results of adapting our process by reducing the fragment size (N) to 5, Table 8 of augmenting the number of hash function used by the minhash generation (K) to 60, and Table 9 of modifying the calculated summaries so that they do not include the location, commentsBefore and commentsAfter attributes. Note that although our prototype provides facilities for filtering out attributes (see the *AttributeFilter* Java class), domain specific knowledge is required to identify such attributes.

All three succeed at augmenting the difference in estimated similarity between very similar (i.e., the original and the slightly mutated transformation) and dissimilar models in the repository while the last one (Table 9) also improves with medium values (the heavily mutated transformation). Note that there is a trade-off

Table 6: Approach Adaptation

Issue	Adaptation
Very Large Model	Increase R and S
Regular Size Model	Reduce N
	Augment K
	Customize Summary
	Customize Fragment Generation

between adapting summaries to take into account certain information and decreasing or increasing the sensibility of the robust hash calculation. Adapting the summaries so that they include different information, in this case, removing location, commentsBefore and commentsAfter attributes reduces spurious differences between all the fragments, and this raises the baseline similarity. At the same time, it gives more weight to real differences which makes the heavily mutated model easier to spot.

7 Application Scenarios

A robust hashing algorithm for MDE artifacts opens the door to efficiently support a number of application scenarios such as search and classification operations, intellectual property protection and authenticity assessment. This section briefly covers some of these applications.

7.1 Classification & Search

We can use the fast comparison and retrieval properties of a robust hashing scheme to detect very similar models in large repositories (such as MDE Forge [34]). This could be useful in diverse scenarios, including: automatic classification of models (e.g., to find all models related to some topic); plagiarism detection in academic assignments, model diversity assessment, etc.

In order to do so, we need to extend our robust hashing algorithm with a *locality-sensitive hashing scheme* aimed at reducing the total amount of comparisons to complete when classifying a large number of models. Even if comparing hashes is much faster than comparing the models themselves, it is still computationally expensive. The goal is to first classify the models to compare into a set of buckets where each bucket holds models that are similar.

The process is very close to our classification step in Section 4, but this time applied to the whole model hash and not to fragments. As in that step, the idea behind this is that most of the dissimilar pairs will never hash to the same bucket (being the number of buckets large enough to avoid accidental collisions).

Once this is done, plagiarism detection gets simplified by reducing it to the problem of comparing the models that ended up in the same bucket. Conversely, model diversity could be ensured by choosing a set of models from different buckets (e.g. to increase coverage in a model-based testing scenario or ease the detection of model variant outliers previous to product-line integration in a variability mining scenario[35]).

Within a bucket we can use standard model comparison tools to directly compare pairwise the models as the amount of comparisons is now drastically reduced. Indeed, LSH is an approximate search mechanism, and thus false positives while rare, may appear. Thus, model comparison and matching tools such as EMFCompare [26] DiffMerge [36], Epsilon Comparison Language [37] or [38] may be used to obtain more accurate results at this point.

In [33] we report on the use of our approach for the case of plagiarism detection for software modeling assignments in model-driven engineering academic programs. We evaluated the feasibility, usefulness and efficiency of using LSH over our robust hashes 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. Therefore, our robust hashing approach enables the integration of plagiarism detection into the toolset and activities of instructors in order to correctly evaluate students and the outputs of MDE programs.

7.2 Copyright Infringement

Robust hashing can be used to provide prosecution evidence in cases of IP protection violations. A model considered to be an unauthorized copy or derivation of a proprietary model would be hashed (using the owner's secret key) and this hash then searched and compared in order to determine its prior existence [9].

For this scenario, the importance of hashing relies not only on its efficient storage (as in the previous section) but also as a way to avoid exposing intellectual property as the models can not be reconstructed from

[9] Hashing may also be used as a key building block in the construction of a watermarking scheme [1] for models but to be most effective the hashing creation process should be different from the one presented here [39].

Table 7: Pairwise Similarity for Model Transformations - B

	a	b	c	d	e	f	g	h	i	j	k	l
a	100	98	38	31	30	34	34	31	32	29	31	32
b	98	100	38	31	30	34	34	31	32	28	31	32
c	38	38	100	33	32	32	30	30	33	30	31	35
d	31	31	33	100	30	31	33	31	32	30	31	32
e	30	30	32	30	100	32	32	29	31	29	30	29
f	34	34	32	31	32	100	33	31	32	30	32	30
g	34	34	30	33	32	33	100	30	33	31	33	31
h	31	31	30	31	29	31	30	100	32	29	31	29
i	32	32	33	32	31	32	33	32	100	31	33	31
j	29	28	30	30	29	30	31	29	31	100	32	31
k	31	31	31	31	30	32	33	31	33	32	100	31
l	32	32	35	32	29	30	31	29	31	31	31	100

*a:tr01_m0.xmi; b:tr01_m1.xmi; c:tr01_m2.xmi; d:tr02.xmi; e:tr03.xmi; f:tr04.xmi; g:tr05.xmi; h:tr06.xmi; i:tr07.xmi; j:tr08.xmi; k:tr09.xmi; l:tr10.xmi;

Table 8: Pairwise Similarity for Model Transformations - C

	a	b	c	d	e	f	g	h	i	j	k	l
a	100	86	34	27	25	25	26	25	27	26	26	28
b	86	100	35	27	24	25	26	26	27	26	26	28
c	34	35	100	26	25	26	25	26	26	25	26	27
d	27	27	26	100	24	25	25	25	27	25	26	27
e	25	24	25	24	100	24	25	24	24	24	26	25
f	25	25	26	25	24	100	25	25	25	24	24	27
g	26	26	25	25	25	25	100	24	26	26	26	26
h	25	26	26	25	24	25	24	100	24	24	24	26
i	27	27	26	27	24	25	26	24	100	26	26	27
j	26	26	25	25	24	24	26	24	26	100	26	27
k	26	26	26	26	26	24	26	24	26	26	100	25
l	28	28	27	27	25	27	26	26	27	27	25	100

*a:tr01_m0.xmi; b:tr01_m1.xmi; c:tr01_m2.xmi; d:tr02.xmi; e:tr03.xmi; f:tr04.xmi; g:tr05.xmi; h:tr06.xmi; i:tr07.xmi; j:tr08.xmi; k:tr09.xmi; l:tr10.xmi;

the hashes (notably, without the secret key). Moreover, the fact that our hashing algorithm is robust is key to detect tampering aimed to avoid copyright infringement detection even for models that are derivations from ours.

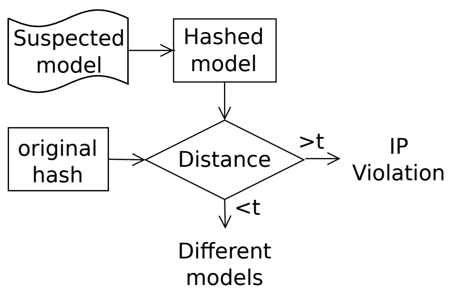


Fig. 12: IP Violation Detection Process

The detection of IP infringements follows the process depicted in Figure 12. Starting from a suspected model, we hash it using our secret key. We then compare it with the hash of the original model to determine if the similarity of the hashes is high enough to determine, within a determined confidence level (this confidence is build as to make statistically very unlikely that such a similarity level arises from independent models), that they are either the same model or derived from each another. Note that in order to efficiently retrieve the original model from its storage, or to proactively find copies across repositories, we can use the classification and search mechanism described in Subsection 7.1.

7.3 Accountability

In a collaborative modeling scenario where a number of different companies participate in the creation and evolution of models, we typically want to keep track of the

Table 9: Pairwise Similarity for Model Transformations - D

	a	b	c	d	e	f	g	h	i	j	k	l
a	100	88	64	41	35	32	33	30	32	36	30	36
b	88	100	69	42	34	33	32	30	31	35	30	36
c	64	69	100	39	32	35	34	30	30	35	31	36
d	41	42	39	100	32	33	31	31	31	31	31	40
e	35	34	32	32	100	34	30	30	29	30	31	31
f	32	33	35	33	34	100	31	30	29	32	32	33
g	33	32	34	31	30	31	100	28	32	36	32	31
h	30	30	30	31	30	30	28	100	31	31	32	33
i	32	31	30	31	29	29	32	31	100	33	34	30
j	36	35	35	31	30	32	36	31	33	100	35	31
k	30	30	31	31	31	32	32	32	34	35	100	34
l	36	36	36	40	31	33	31	33	30	31	34	100

*a:tr01_m0.xmi; b:tr01_m1.xmi; c:tr01_m2.xmi; d:tr02.xmi; e:tr03.xmi; f:tr04.xmi; g:tr05.xmi; h:tr06.xmi; i:tr07.xmi; j:tr08.xmi; k:tr09.xmi; l:tr10.xmi;

changes performed by each party and the corresponding model versions generated at each step. This information could be, for instance, stored in a blockchain infrastructure.

The benefits of the use of blockchain technologies as a decentralized consensus ledger to provide accountability for the actions of different agents on shared resources have been already acknowledged in different application domains [40] [41]. However, its adaptation to the MDE environment risks to present scalability issues. Indeed, directly storing full models as part of blockchain transactions would fail to scale as models are too large for what current blockchain infrastructure can efficiently handle nowadays.

In this sense, our hashing mechanism for models may become a key enabler for the use of blockchain technologies for the enforcement of accountability in collaborative development scenarios by remarkably reducing the required storage requirements if we store the hash, instead of the full model, in the blockchain transactions. Similarly, looking for all transactions pertaining to a given model would be easier and quicker to perform by using the model digests provided by our hashing algorithm instead of their full model counterparts.

8 Related Work

Robust hashing algorithms for different digital assets have attracted a great deal of attention from the research community over the last decade. To the best of our knowledge, our approach is the first robust hashing algorithm specifically designed for MDE artefacts.

Notably many different robust hash algorithms have been proposed for the domains of digital image [1] [42] [43] [44] [45], 3D mesh models [2] [46], digital video [47] [48] and text [3].

While the general, high-level process for generating the robust hashes is similar across domains, the concrete steps need to be adapted to each domain and to the desired robustness (e.g., the kind of modifications to be resisted). As such, techniques for the other domains cannot be directly applied to the MDE domain.

We base our approach on the minhash [7] and locality sensitive hashing [18] techniques. Minhash has been used before on model-like structures. In [39] in order to provide a robust labeling mechanism for model elements that enables the use of state-of-the-art watermarking algorithms in an MDE ecosystem and in [49] in order to match equivalent terms in different ontologies. This latter approach works at the individual class / term level and therefore cannot be used at the global model level (e.g., structure is not taken into account). As for the watermarking algorithms in [39], they could be used to extract patterns (instead of performing insertions) to be used as hashes, but those hashes would be less effective as model *digests* than the ones we generate here as they were created with a different and very application-specific goal in mind (the protection of the watermark).

Prior to us, text analysis techniques for the purpose of model comparison have been used in [50], where the authors use Natural Language Processing (NLP) to find the semantic similarities between language descriptions by attaching text descriptions to domain concepts. We see their approach as complementary to ours, since these annotations could be added to the models before the hashing phase in order to take into account model semantics in our approach. Similarly, in [51] and [52] the authors use 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 [53]) for the purposes of structural model clustering and clone detection. Again, we believe their approach, as other model matching, comparison and clone detection contributions, may be an interesting complementary technique to ours for specific use cases such as plagiarism detection (e.g., see our own contribution on the subject [33]). Indeed, those techniques can be applied in a second phase, once plagiarism candidates have been found with our approach in order to find the concrete copied parts.

MAR [32], a structure-based search engine for models, represents models as a bag of (configurable) paths extracted from a corresponding multigraph representation. Our robust hashing approach may be seen as an indexing alternative while offering as advantages a higher robustness and smaller size.

Finally, similarly to us, in [27] and [28] the authors describe a vectorization approach for models followed by the calculation of distance metrics over these vectors. However, their vector extraction approach is very different to ours. In their approach, a vector from a model is built by concatenating (numeric) data related with attributes and links on each instance (i.e., node). Unlike ours, their approach does not include mechanisms to enhance the robustness and the compactness of the extracted vector such as fragment extraction and minhash calculation.

9 Conclusions and Future Work

In this paper, we have explored the adaptation of the robust hashing concept to the modeling domain. We have shown how the adaptation of diverse techniques such as summary extraction, minhash generation and locality-sensitive hash function families, originally developed for the comparison and classification of large repositories of data, can be used to effectively extract the core features of a model to produce a model-based robust hash.

We have provided a prototype implementation of our approach that we have used to show that: 1) our approach can deal with any graph-based model representation; 2) a strong correlation exists between the similarity calculated directly on the robust hashes and a distance metric calculated over the original models; 3) our approach scales well on large models and greatly reduces the time required to find similar models in large repositories. As such, we believe our robust hashing algorithm can become a key building block in any solution aimed at intellectual property protection, authenticity assessment and fast comparison and retrieval, all of them key requirements for the adoption of model-driven engineering in complex industrial environments.

In the future, we plan to continue this line of work by looking into the following directions of further work. First, we would like to study the benefits of developing specific adaptations of the approach tailored to concrete types of models (and therefore using the specific semantics of that model type in the hashing strategy) for those concrete scenarios where this fine tuning may be necessary. Second, we plan to extend our approach to cover the hashing of sets of interrelated models [54]. Finally, we will study the benefits and potential of combining robust hashing with a blockchain for models infrastructure (where hashes could be used to efficiently track model transactions on the blockchain when there is no need to store the full model) and with machine learning techniques in order to automatically derive good hash functions for specific model datasets [55]. In that sense, we also intend to explore the use of existing consistency preserving model slicers as and alternative to our generic fragmentation process [56, 57].

References

1. Jiri Fridrich and Miroslav Goljan. Robust hash functions for digital watermarking. In *Information Technology: Coding and Computing, 2000. Proceedings. International Conference on*, pages 178–183. IEEE, 2000.
2. Suk-Hwan Lee and Ki-Ryong Kwon. Robust 3d mesh model hashing based on feature object. *Digital Signal Processing*, 22(5):744–759, 2012.
3. Martin Steinebach, Peter Klöckner, Nils Reimers, Dominik Wienand, and Patrick Wolf. *Robust Hash Algorithms for Text*, pages 135–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-40779-6.
4. Ronald Rivest. The md5 message-digest algorithm. 1992.
5. D Eastlake 3rd and Paul Jones. Us secure hash algorithm 1 (sha1). Technical report, 2001.
6. Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228(5):15–23, 1973.
7. Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.
8. Salvador Martínez, Sébastien Gérard, and Jordi Cabot. Robust hashing for models. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 312–322, 2018.

9. David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.
10. Eugene Syriani, Robert Bill, and Manuel Wimmer. Domain-specific model distance measures. *Journal of Object Technology*, 18(3), 2019.
11. Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.
12. Kevin Lano and Shekoufeh Kolahdouz Rahimi. Slicing techniques for uml models. *Journal of Object Technology*, 10(11):1–49, 2011.
13. Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In *International Conference on Model Driven Engineering Languages and Systems*, pages 62–76. Springer, 2011.
14. Daniel Struber, Julia Rubin, Gabriele Taentzer, and Marsha Chechik. Splitting models using information retrieval and model crawling techniques. In *International Conference on Fundamental Approaches to Software Engineering*, pages 47–62. Springer, 2014.
15. Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Software R eliability Engineering, 2006. ISSRE'06. 17th International Symposium on*, pages 85–94. IEEE, 2006.
16. Markus Scheidgen. Reference representation techniques for large models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 5. ACM, 2013.
17. Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. Model composition-a signature-based approach. In *Aspect Oriented Modeling (AOM) Workshop*, 2005.
18. Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014.
19. Ari Juels and Martin Wattenberg. A fuzzy commitment scheme. In *Proceedings of the 6th ACM conference on Computer and communications security*, pages 28–36. ACM, 1999.
20. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.
21. Javier Troya, Martin Fleck, Marouanne Kessentini, Manuel Wimmer, and Bader Alkhaze. Rules and helpers dependencies in atl–technical report. *Universidad de Sevilla*, 2016.
22. Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. Adaptability of model comparison tools. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 306–309. IEEE, 2012.
23. Cody Kinneer and Sebastian JI Herzig. Dissimilarity measures for clustering space mission architectures. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 392–402, 2018.
24. Oszkár Semeráth, Rebeka Farkas, Gábor Bergmann, and Dániel Varró. Diversity of graph models and graph generators in mutation testing. *International Journal on Software Tools for Technology Transfer*, 22(1):57–78, 2020.
25. Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. Different models for model matching: An analysis of approaches to support model differencing. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6. IEEE, 2009.
26. Cédric Brun and Alfonso Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.
27. Adel Ferdjoukh, Florian Galinier, Eric Bourreau, Annie Chateau, and Clémentine Nebut. Measuring differences to compare sets of models and improve diversity in mde. In *ICSEA: International Conference on Software Engineering Advances*, 2017.
28. Adel Ferdjoukh, Florian Galinier, Eric Bourreau, Annie Chateau, and Clémentine Nebut. Measurement and generation of diversity and meaningfulness in model driven engineering. 2018.
29. Antoine Toulmé and I Inc. Presentation of emf compare utility. In *Eclipse Modeling Symposium*, pages 1–8, 2006.
30. Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *European Conference on Object-Oriented Programming*, pages 600–624. Springer, 2007.
31. Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, page 1, 2001.
32. José Antonio Hernández López and Jesús Sánchez Cuadrado. Mar: a structure-based search engine for models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 57–67, 2020.

33. Salvador Martínez, Manuel Wimmer, and Jordi Cabot. Efficient plagiarism detection for software modeling assignments. *Computer Science Education*, pages 1–29, 2020.
34. Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. Mdeforge: an extensible web-based modeling platform. In *CloudMDE@ MoDELS*, pages 66–75, 2014.
35. David Wille, Önder Babur, Loek Cleophas, Christoph Seidl, Mark van den Brand, and Ina Schaefer. Improving custom-tailored variability mining using outlier and cluster detection. *Science of Computer Programming*, 163:62–84, 2018.
36. O Constant. Emf diff/merge, 2012.
37. Dimitrios S Kolovos. Establishing correspondences between models with the epsilon comparison language. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 146–157. Springer, 2009.
38. Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel matching for automatic model transformation generation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 326–340. Springer, 2008.
39. Salvador Martínez, Sébastien Gérard, and Jordi Cabot. On watermarking for collaborative model-driven engineering. *IEEE Access*, 6:29715–29728, 2018.
40. Fernando Gomes Papi, Jomi Fred Hübner, and Maiquel de Brito. Instrumenting accountability in mas with blockchain. *Accountability and Responsibility in Multiagent Systems*, page 20.
41. Ricardo Neisse, Gary Steri, and Igor Nai-Fovino. A blockchain-based approach for data accountability and provenance tracking. *arXiv preprint arXiv:1706.04507*, 2017.
42. Ram Kumar Karsh, RH Laskar, and Bhanu Bhai Richhariya. Robust image hashing using ring partition-pgnmf and local features. *SpringerPlus*, 5(1):1995, 2016.
43. YuLing Liu and Yong Xiao. A robust image hashing algorithm resistant against geometrical attacks. *Radio Eng*, 22(4):1072–1081, 2013.
44. Ashwin Swaminathan, Yinian Mao, and Min Wu. Robust and secure image hashing. *IEEE Transactions on Information Forensics and security*, 1(2): 215–230, 2006.
45. Ramarathnam Venkatesan, S-M Koon, Mariusz H Jakubowski, and Pierre Moulin. Robust image hashing. In *Image Processing, 2000. Proceedings. 2000 International Conference on*, volume 3, pages 664–666. IEEE, 2000.
46. Khaled Tarmissi and A Ben Hamza. Information-theoretic hashing of 3d objects using spectral graph theory. *Expert Systems with Applications*, 36(5): 9409–9414, 2009.
47. Baris Coskun and Bulent Sankur. Robust video hash extraction. In *Signal Processing Conference, 2004 12th European*, pages 2295–2298. IEEE, 2004.
48. Cedric De Roover, Christophe De Vleeschouwer, Frédéric Lefebvre, and Benoit Macq. Robust video hashing based on radial projections of key frames. *IEEE Transactions on Signal processing*, 53(10): 4020–4037, 2005.
49. Michael Cochez. 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)*, pages 134–140. IEEE, 2014.
50. Florian Noyrit, Sébastien Gérard, and François Terrier. Computer assisted integration of domain-specific modeling languages using text analysis techniques. In *International Conference on Model Driven Engineering Languages and Systems*, pages 505–521. Springer, 2013.
51. Önder Babur and Loek Cleophas. Using n-grams for the automated clustering of structural models. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 510–524. Springer, 2017.
52. Önder Babur, Loek Cleophas, and Mark van den Brand. Metamodel Clone Detection with SAMOS. *Journal of Computer Languages*, 51:57 – 74, 2019.
53. William B Cavnar, John M Trenkle, et al. N-gram-based text categorization. In *Proceedings of the 3rd Symposium on Document Analysis and Information Retrieval (SDAIR)*, 1994.
54. Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.
55. Jingkuan Song, Yi Yang, Xuelong Li, Zi Huang, and Yang Yang. Robust hashing with local models for approximate similarity search. *IEEE transactions on cybernetics*, 44(7):1225–1236, 2014.
56. Christopher Pietsch, Manuel Ohrndorf, Udo Kelter, and Timo Kehrer. Incrementally slicing editable submodels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 913–918. IEEE, 2017.

57. Gabriele Taentzer, Timo Kehrer, Christopher Pietsch, and Udo Kelter. A formal framework for incremental model slicing. In *International Conference on Fundamental Approaches to Software Engineering*, pages 3–20. Springer, Cham, 2018.