Metamodel clone detection with SAMOS

Citation for published version (APA):

Babur, Ö., Cleophas, L., & van den Brand, M. (2019). Metamodel clone detection with SAMOS. *Journal of Computer Languages*, 51, 57-74. <https://doi.org/10.1016/j.col.2018.12.002>

Document license:

TAVERNE

DOI:

[10.1016/j.col.2018.12.002](https://doi.org/10.1016/j.col.2018.12.002)

Document status and date:

Published: 01/04/2019

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Metamodel clone detection with SAMOS

Önder Babur^{a,*}, Loek Cleophas^{a,b}, Mark van den Brand^a

^a Department of Mathematics and Computer Science, Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands

^b Department of Information Science, Stellenbosch University, 7602 Matieland, South Africa

ARTICLE INFO

Keywords:

Model-driven engineering
Domain-specific languages
Model analytics
Model clone detection
Vector space model
Clustering
Repository mining
Software maintenance
Empirical software engineering

ABSTRACT

Wider adoption of model-driven engineering leads to an abundance of models and metamodels in academic and industrial practice. One of the key techniques for the management and maintenance of such artifacts is model clone detection, where highly similar (meta-)models and (meta-)model fragments are mined from a typically large set of data. In this paper we have extended the SAMOS framework (Statistical Analysis of Models) to clone detection, exemplified on Ecore metamodels. Our clone detection approach uses and extends the framework's feature extraction, vector space model, natural language processing and clustering capabilities. We performed three extensive case studies to demonstrate its accuracy both quantitatively and qualitatively. We first compared the sensitivity and accuracy of SAMOS for metamodel changes through mutation and scenario analysis (which simulate clones) with those of NICAD-Ecore and MACH, tools for clone detection on Ecore and UML models respectively. We then compared the precision and recall of SAMOS and of NICAD-Ecore on a real dataset, consisting of conference management metamodels from the ATL Zoo. Finally we performed a repository-wide mining of metamodel clones from GitHub. We conclude that SAMOS stands out with its higher accuracy and yet considerable scalability for further large-scale clone detection and other empirical studies on metamodels and domain specific languages.

1. Introduction

Model-driven engineering (MDE) promotes the use of models (and metamodels to which they conform) as central artifacts in the software development process¹. This eases development and maintenance of software artifacts (including source code generated from models), yet increasing MDE adoption leads to an abundance of models in use. Some examples of this include the academic efforts to gather models in repositories, and large-scale MDE practices in the industry [1–3]. This leads to challenges in the management and maintenance of those artifacts. One of those challenges is the identification of model clones, which can be defined in the most general sense as duplicate or highly similar models and model fragments [4]. Similar scenarios apply in the traditional software development for source code clones. There is a significant volume of research on code clones, elaborating the drawbacks of having clones, which can be a major source of defects or lead to higher maintenance cost and less reusability, and providing detection techniques and tools [5]. Note that in some cases clones might be useful too, as argued by Kapser and Godfrey [6]; it is nevertheless worthwhile to investigate them. Code clones have attracted the attention of the source code analysis community, who had to deal with the maintenance

of large numbers of artifacts for a longer time than the MDE community.

Model clone detection, on the other hand, is a relatively new topic. Many researchers have drawn parallels from code clones, and claimed that a lot of the issues there can be directly translated into the world of models. While the problem domains are similar, the solution proves to be a challenge. Source code clone detection usually works on linear text or an abstract syntax tree of the code while models are in general graphs [7]; other aspects are also inherently different for models, such as tool-specific representations, internal identifiers, and abstract vs. concrete syntaxes [8].

There are several approaches for model clone detection, or model comparison in the broader sense, in the literature [7,9]; yet we are particularly interested in ones with a publicly available tool to be used in our studies. A good portion of such tools are either limited to, tailored for, or evaluated on specific types of models such as MATLAB/Simulink. Notable examples, along with the code clone detector backends for them, are CloneDetective-ConQAT [4] and the SIMONE Simulink clone detector [10]—referred to as NICAD-SIMONE in this paper—an extension of text-based tool NICAD to cover clone detection in Simulink models. Another interesting approach for Simulink models is

* Corresponding author.

E-mail addresses: O.Babur@tue.nl (Ö. Babur), L.G.W.A.Cleophas@tue.nl (L. Cleophas), M.G.J.v.d.Brand@tue.nl (M. van den Brand).

¹ In our context, we refer to metamodels and models shortly as *models*, as metamodels are models too.

ModelCD based on graph comparison and approximation [11], but the tool is not publicly available. While the approach is similar to ours in this paper, given the unavailability we cannot use it in our study; see however Section 8 for a discussion. Störrle presents an approach and a tool, *MQ_{clone}*, for UML model clone detection [8,12]. Störrle elaborately describes and classifies UML model clones, noting the differences to code clones and Simulink model clones. Furthermore, the author reports a much higher performance and scalability for *MQ_{clone}* compared to ConQAT and ModelCD. *MQ_{clone}* is integrated into the publicly available tool suite MACH,² though in a very limited manner, with almost no control over the rich set of algorithms and settings developed by Störrle.

In our research we have the goal of detecting clones in large repositories of models and large evolving industrial domain-specific language (DSL) ecosystems based on the Eclipse Modelling Framework (EMF). Metamodels are artifacts of particular interest to us, for various purposes including metamodel repository management and DSL analysis (see Section 2 for a detailed discussion). To achieve this goal, we have investigated the feasibility of existing tools, with three major requirements: (1) conceptual and technological applicability to Ecore metamodel clones; (2) sensitivity to all possible metamodel changes, and accuracy in general (precision, recall); and (3) scalability for large datasets. As a starting point we considered MACH and NICAD-SIMONE as promising candidates. However, these tools underperformed with respect to some of our requirements, which will be demonstrated in the rest of this paper.

We have taken an orthogonal approach by extending the SAMOS framework (Statistical Analysis of MOdels) for model clone detection. SAMOS is a state-of-the-art tool for large-scale analysis of models [13]. We wish to exploit the underlying capabilities of the framework—incorporating information retrieval-based fragmentation, natural language processing, and statistical algorithms—for model clone detection. In this paper, we describe how we have extended and tailored SAMOS for (meta)model clone detection. This paper greatly extends our previous work [14] with new techniques (notably tree extraction and comparison) and an extensive quantitative and qualitative evaluation of its accuracy in multiple case studies; including comparison with MACH, and with NICAD-Ecore, our custom NICAD extension for Ecore which was inspired by NICAD-SIMONE.

The rest of the paper is structured as follows. Section 2 opens with scenarios and conceptualization of metamodel clones. We outline the two clone detector tools used to contrast SAMOS to in our case studies, namely NICAD-ECORE and MACH, in Section 3. The background concepts from information retrieval that inspired and formed a basis for our approach are introduced in Section 4. In Section 5, we elaborate on the extended SAMOS framework with its feature extraction, comparison, natural language processing and clustering capabilities. Section 6 details the three extensive case studies with mutation/scenario analysis for SAMOS, NICAD-Ecore and MACH; comparison of SAMOS with NICAD-Ecore on ATL Zoo metamodels; and finally a repository mining scenario on a very large set of GitHub metamodels. The rest follows with an overall discussion including future work in Section 7 and further related work in Section 8. We finally draw conclusions on the applicability of SAMOS to metamodel clone detection.

2. Metamodel clones

The goal of our research is to detect metamodel clones. Metamodel clones might exist due to a wide range of reasons including copy-paste or clone-and-own approaches in model-driven development [7], lack of abstraction mechanisms in metamodels for language design [15], or difficulty in reuse for DSLs in general [16]. Maintenance, which has been identified by Kosar et al. [17] as one of the major overlooked areas

in DSL research, is hampered by the presence of clones and can benefit from clone detection [18].

In our research, we are interested in finding similar (fragments of) metamodels, with the following problems at hand. First and foremost, clones suggest potential scenarios for quality assurance and refactoring in MDE/DSL ecosystems. Also, as those ecosystems in large-scale settings do consist of multiple DSLs, clone detection across different DSLs and their versions forms a basis of empirical studies on their development and evolution. Furthermore, repositories and datasets of metamodels, whether company-wide in industry, online in the public domain, or specific collections for research purposes, could benefit from clone detection for activities such as repository management, exploration, data preprocessing, filtering, and large-scale empirical studies (such as on the origin, distribution and genealogy of the clones). Finally, we might use clone detection for plagiarism detection and assistance in grading student assignments for language design and metamodeling courses, similarly as done by the counterparts in the source code domain, such as JPLag [19].

While metamodel clones have not been specifically studied in the literature, model clones have been, particularly model clones in MATLAB/Simulink and other data flow type models. A very large portion of the model clone detection literature is focused on data flow languages such as Simulink with the following classification scheme [10,20]:

- *Type-I* (exact) model clones: Identical model fragments except for variations in visual presentation, layout and formatting.
- *Type-II* (blind renamed, or consistently renamed) model clones: Structurally identical model fragments except for variations in labels, values, types, visual presentation, layout and formatting.
- *Type-III* (near-miss) clones: Model fragments with further modifications, such as changes in position or connection with respect to other model fragments and small additions or removals of blocks or lines in addition to variations in labels, values, types, visual presentation, layout and formatting.
- *Type-IV* (semantic) clones: Model fragments with different structure but equivalent or similar behaviour.

Note that we omit Type-IV, i.e. semantic, clones for the scope of this paper as they pose an orthogonal and arguably bigger challenge; however this is perfectly in line with the related work for NICAD-SIMONE and MACH, both of which omit semantic clones as well.

While the above scheme is more or less a community standard, metamodels are more similar to UML class diagrams than to Simulink models, with respect to two key aspects of (1) the importance of the model element names, and (2) the dominance of a containment tree structure. Hence our conceptualization and classification of metamodel clones is mostly adopted from [12]. Störrle emphasizes that names of model elements are essential parts of UML models. His classification for UML clones has a notable distinction from the Simulink classification: it rules out Type-II (renamed) clones due to the indispensability of element names. This is also the case in the context of EMF metamodel clones. Our clone classification, adding a few items related to Natural Language Processing (NLP) to Störrle's classification, is given below. He further argues that the structure of UML models is dominated by a containment tree with few additional cross-tree connections. We believe those two observations apply to Ecore metamodels as well; though an empirical study to find evidence on large corpora is left as future work. Note we use a different formulation than of Störrle for semantic clones. However, we omit Type-D semantic clones for the scope of this paper (see discussion above for Type-IV clones), which is consistent with Störrle who avoids semantic clones as well. We leave it as future work to conceptualize and detect semantic model clones. We will use the following classification for the rest of the paper.

- *Type-A* duplicate model fragments except secondary notation

² <https://www.pst.ifi.lmu.de/~stoerrle/tools/mach.html>.

(layout, formatting), internal identifiers.

- Plus *any cosmetic change* in the names (lower-/uppercase, snake-/camel-case and other insignificant changes).
- *Type-B* duplicate model fragments with *small percentage* of changes to names, types, attributes, few additions/removals of parts.
 - Plus *potentially many syntactic/semantic changes* in the names such as typos, synonyms, semantically related words.
- *Type-C* duplicate model fragments with *substantial percentage* of changes/ additions/removals of names, types, attributes and parts.
- *Type-D* semantically equivalent or similar model fragments with different structure and content.

Now that we have set the concepts and classification of metamodel clones, we continue with existing model clone detectors that can be used for metamodel clone detection.

3. Other model clone detector tools

In this section we discuss two prominent model clone detector tools we used in our comparative evaluation against SAMOS.

3.1. NICAD-SIMONE

The NICAD Clone Detector³ is a scalable clone detection tool implementing the NICAD (Automated Detection of Near-Miss Intentional Clones) code clone detection method [21]. It was mainly designed for finding intentionally copy/pasted units, such as functions and sub-systems, that have been modified. It uses a configuration file to specify steps such as normalization of identifier names and filtering of irrelevant parts. It has good reporting capabilities in XML and HTML for readability. NICAD supports a range of languages and normalizations, and is designed to be easily extensible using a component-based plug-in architecture. Furthermore, it is scalable to very large systems with millions of lines of code.

Internally, NICAD is a parser-based and language-specific but lightweight tool, which adopts a line-based textual comparison rather than subtree comparison to achieve better performance and scalability. It is built on top of the TXL (short for Turing eXtender Language) programming and transformation language [22] for identifying syntactic clones while relying on pretty-printing to eliminate formatting differences and noise. Thanks to TXL, the tool is extensible to other languages via the introduction of the appropriate TXL grammar and transformation rules [23].

For the scope of this paper, the main feature of interest is the standard mode of operation in the model clone detection tools derived from NICAD (notably SIMONE [10]). The approach consists of (1) parsing textual forms of models which may use various formatting and preprinting, (2) pre-processing to a normalized format, removing irrelevant parts, and ordering textual elements using multi-attribute topological sorting, (3) extracting subparts with the selected granularity for the comparisons such as subsystem scope for Simulink, (4) post-processing to further normalize, for instance rename identifier names, filter or transform elements, (5) line-based computation of least common subsequence (LCS) to find clone pairs, (6) clustering the pairs using connected component analysis.

There are two points to discuss in detail about this approach. For the LCS algorithm, the order of the elements naturally matters. Step (2) is needed to properly normalize reordering of elements because different orderings of the same elements still represent clones such as with data-independent declaration statements in code [24] or certain elements in models (UML models [8], Simulink models [10], metamodels [14]). For instance, NICAD-SIMONE indeed underlines a fundamental problem of line-based comparison on graphical data: the order of (at least some of

the) elements actually does not lead to any meaningful change. NICAD-SIMONE tackles this by a sorting-normalization pass where it sorts model elements according to their type, name and so on, and therefore can successfully detect clones with reordered elements. Another point is the line-based comparison in LCS, which is done in NICAD outside the TXL scope (in a Turing + script) and therefore hard to do in a language-aware manner. In our experience this line-based approach is followed consistently in NICAD-based approaches, whether for model clone detection as by Antony et al. for UML sequence diagrams [25] or code clone detection.⁴

3.1.1. Extending NICAD for metamodel clone detection

At Eindhoven University of Technology, we have had a small project for an Ecore extension for metamodel clone detection. The base extension can be found in GitHub.⁵ It is inspired by NICAD-SIMONE, i.e. the extension of NICAD for Simulink models, and tries to replicate the same mode of operation for detecting Ecore metamodel clones. It has the following major steps for its workflow:

1. Convert Ecore metamodels into the textual Emfatic⁶ format,
2. Provide an Emfatic TXL grammar for NICAD,
3. Provide an Ecore plug-in with transformation rules for extraction, filtering, renaming and sorting of model elements.

In order to do a correct clone detection and fair comparison with other tools, we have developed an improved extension of this plug-in (replicating the mode of operation of NICAD-SIMONE as close as possible) with the following features:

- *Granularity*. Allow extraction with two fixed scopes: whole-model, or EClass (a metaclass of Ecore, similar to Class in UML class diagrams),
- *Filtering*. Remove most of the original filtering and simplifications (as implemented by the student), to keep as much information as possible for completeness, still excluding elements which SAMOS ignores such as EAnnotations (a metaclass of Ecore for annotations, see Section 5.3 for details),
- *Sorting*. Implement multi-attribute topological sorting wherever applicable (such as among all elements contained in an EClass, multiple supertypes and so on) with respect to type, eType and name of the corresponding elements,
- *Configuration*. Add configuration files to detect Type A, B and C clones with different threshold values; respectively 0, 0.1 and 0.3.

3.2. MACH

The MACH toolset for UML model analysis and checking includes the UML model clone detector MQ_{clone} developed by Störrle [8,12]. The approach is applicable for multiple types of UML models, ranging from class diagrams to use case and sequence diagrams; in contrast to several existing approaches focusing on a single type of model, such as MATLAB/Simulink models [11]. The most relevant steps of the approach from the perspective of this paper are (1) processing (XMI-serializations of) models as graphs, and encoding the graphs as Prolog programs representing node and edge information; (2) defining various similarity heuristics based on element names or hash/index values; and finally (3) using Prolog rules and inference to match, rank, and weight clones in the models. MACH natively supports UML files produced by tools such as MagicDraw; and automatically eliminates/normalizes layout, internal identifiers and tool-specific information.

⁴ See a small Java code exercise in our report <http://www.win.tue.nl/~obabur/publications/JVLC18/>.

⁵ <https://github.com/jzhang3/Nicad3-emfatic>.

⁶ <https://www.eclipse.org/emfatic/>.

³ <https://www.txl.ca/nicaddownload.html>.

The author advocates focusing on node similarity rather than the graph structure similarity (as done in other tools such as [4,26]); and claims that UML models are very wide and flat trees with mostly containment structure in contrast to more general graphs. The clone detection is based on a variety of heuristics. First, name-based similarity heuristics involve comparing element names based on the Levenshtein edit distance and taking into account other factors such as different format, for example CamelCaps vs. separate words. Such heuristics include one which takes just the name of a single node for comparison, and another which also considers the neighbouring nodes' names for similarity. In [12], a new heuristic is introduced where type and attribute information is also taken into account. Another set of index-based heuristics involve comparing the hash values of model elements: for instance, using a hash function which adds up all characters found in two model elements. Finally, there is a more powerful similarity heuristic which can weight and rank clones, and hence favors more promising matches as clone candidates.

An important note is that the rich set of heuristics and settings developed by Störle is only partially accessible in the closed source MACH toolset. The single standard clone detection setting in MACH calculates similarities with absolute measures, rather than standardized in the range of [0,1] or in percentages – which makes it harder to interpret the similarity. Furthermore, MACH finds clone pairs but does not group clones into larger clusters.

4. Preliminaries: information retrieval, vector space model, clustering

We discuss here the underlying concepts of the SAMOS framework [13,27] inspired by information retrieval (IR) and machine learning (ML). IR deals with effectively indexing, analyzing, searching and comparing various forms of content including natural language text documents [28]. As a first step for document retrieval in general, documents are collected and indexed via some unit of representation (*vocabulary* in short): this unit can be bag of words (simply all words, all except stop words, or only some domain-specific terms of interest). Alternatively more complex constructs can be used such as n-grams. N-grams originate from computational linguistics and represent a linear encoding of (text) structure, for example “Julius Caesar” as a single entity rather than each word separately.

Index construction can be implemented using a vector space model (VSM) with the following major components: (1) a vector representation of occurrence or frequency of the vocabulary in a document (named *term frequency* for the latter), (2) optionally *zones* (for instance ‘author’ or ‘title’), (3) weighting schemes such as inverse document frequency (idf), and zone weights, (4) NLP techniques for handling compound terms, detecting synonyms and semantically related words.

As an example, Manning et al. present a very simplistic representation of Shakespeare’s plays (originally intended for boolean retrieval) [28]: the vocabulary consists of some important terms, and the vector space is populated with the incidence (i.e. not the frequency) of those terms in the respective plays. Table 1 depicts an excerpt from

Table 1

A simple term incidence matrix representation of Shakespeare’s plays (excerpt from [28]).

	Anthony and cleopatra	Julius caesar	The tempest	Hamlet	Othello	Macbeth	...
Anthony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
...							

Manning’s example.

The whole process described in this section is what we refer to as a traditional workflow of document clustering in IR.

As exemplified, the VSM allows transforming each document into an n -dimensional vector, thus resulting in an $m \times n$ matrix for m documents. Over the VSM, document similarity can be defined as the distance, such as Manhattan, Euclidean or cosine, between vectors. Following the example, a very simple dot product of the VSM in Table 1 would give us the $m \times m$ pair-wise distance matrix, where (computed for the limited example shown in the table) for instance “Anthony and Cleopatra” · “Julius Caesar” = 3, “Anthony and Cleopatra” · “Othello” = 1 and “Anthony and Cleopatra” · “The Tempest” = 0.

Such a distance matrix can in turn be used for identifying similar groups of documents via an unsupervised ML technique called clustering [28,29]. Two commonly used techniques for this are k-means and hierarchical clustering. k-means is a simple but effective one, which aims to identify cluster centres and minimizes the residual sum of (squares of) distances of the points assigned in each cluster. Hierarchical clustering techniques assume an unknown number of clusters and rather build a nested tree structure (*dendrogram*) of the data points, representing proximities and potential clusters.

5. Using and extending SAMOS for clone detection

The framework SAMOS (Statistical Analysis of Models) is a state-of-the-art tool for large-scale analysis of models [13]. It treats models as data, i.e. as documents in IR parlance, and applies document clustering (as introduced in Section 4) to models, starting with a metamodel-driven extraction of features. Features can be, for instance, singleton names of model elements (very similar to the vocabulary of documents) or larger fragments of the underlying graph structure such as n -grams [30]. In our context, an example n -gram for a EMF metamodel would be for $n = 2$ an EClass containing an EAttribute [31]. SAMOS computes a term-frequency based VSM, using comparison schemes (for instance determining whether to match metaclasses or ignore them), weighting schemes (for instance EClass weighted higher than EAttribute) and NLP such as stemming and synonym checking. Note that this is much more sophisticated and capable than the term incidence matrix presented in Section 4. Applying various distance measures suitable to the problem at hand, SAMOS applies different clustering algorithms (via the R statistical software [32]) and can output automatically derived cluster labels or diagrams for visualization and manual inspection and exploration. Fig. 1 illustrates the workflow, with key steps of the workflow and several application areas including domain analysis and clone detection. We concisely give an example-driven traversal of this workflow (with simple settings) in Section 5.1, for a more detailed version please refer to our previous work [27].

5.1. An example run of SAMOS

We start with four metamodels related to state machines, obtained by taking top results when we searched GitHub for “state machine”.e-core files [27]. The dataset, depicted in Fig. 2, consists of two plain finite state machine (FSM) metamodels; one hierarchical FSM metamodel (the latter has the package name FSM though); and one data flow metamodel.

From the input metamodels and Ecore meta-metamodel, we construct the vocabulary in the form of type-name pairs. We adopt a *bag of words* representation for this vocabulary, where each item in the vocabulary is considered individually, discarding context and order. The (domain) type information comes from Ecore metaclasses (such as EPackage and EClass). Next, we recursively go over the content for each metamodel element to extract the vocabulary. The first metamodel in Fig. 2 would yield *Metamodel 1* = {(EPackage, FSM), (EClass, StateMachine), (EReference, transitions),...}. As identifiers in the metamodels typically are compound names, we apply tokenization, stop-word

Fig. 1. Overview of SAMOS workflow.

removal and normalization, for instance, into lower case, to convert identifiers into their tokens as the vocabulary. For checking word similarity, we employ another array of techniques including stemming, typo checking and semantic relatedness/synonymy.

We further use a relaxed type matching (allowing partial score for non-matching types) and synonym matching scheme. As an example a type-name pair ($EAttribute, name$) would yield 1 when matched against itself, while yielding $0.5 \cdot 0.88 = 0.44$ against ($EReference, label$), where 0.88 is the WordNet similarity score of 'name' and 'label' and 0.5 is a multiplier applied because the types do not match. With further weighting schemes (see supplemental material for full list of weights⁷), we calculate a simple term incidence matrix, i.e. our VSM. A part of this table is given in Table 2.

As the next step of SAMOS, we reduce the metamodel similarity problem into a distance measurement of the corresponding vector representations of metamodels. We pick cosine distance as the problem scenario resembles the *topic-wise relatedness* setting in traditional IR. To realize the hierarchical clustering, we calculate the pairwise distance matrix of all the metamodels. The distance matrix for the example dataset is given in Table 3.

We apply agglomerative hierarchical clustering over the VSM to obtain a dendrogram visualization. The interpretation of this diagram depicted in Fig. 3 is as follows: the red and green dotted line at heights 0.3 and 0.6 (manually inserted by us) denote horizontal cuts in the dendrogram. Metamodel 4, which stays far above the cut, can be considered as a clear outlier. Depending on the requirements and interpretation of the user, Metamodels 1–3 can be considered to be in one single cluster (i.e. dendrogram cut at height = 0.6) or just Metamodels 2 and 3 (i.e. cut at height = 0.3).

Note that this is a very simplistic setting of SAMOS. We have used and greatly extended various features of SAMOS, in order to apply it to metamodel clone detection. In the rest of this section, we elaborate the extensions, compartmentalized as key steps of the workflow.

5.2. Scoping/granularity for extraction

While, originally, SAMOS handles entire models and extracts all the model elements contained, we introduce the notion of scoping to define the granularity of independent data elements. For Ecore metamodels, we define three scopes: the whole model, EPackage, and EClass; it is relatively straightforward to pick these fixed scopes as the major *hubs* in the containment tree. The scope guides the extraction by mapping a model into one (i.e. whole model) or more data points such as per EPackage contained. Given a fixed scope, we adopt the approach for UML clone detection [12]: we cover all the model elements under transitive containment closure of that starting model element.

5.3. Extracting model element information

The main unit of information extracted by SAMOS has previously been the so called *type-name* pair, which essentially maps to a vertex in the underlying graph of the model. Such a pair encodes the domain-specific type (metaclass) information, such as EClass, and the name, such as Book, of a model element. For proper clone detection, we need to cover the attributes in the model elements (for instance whether an EClass is abstract) and cardinalities, e.g. of EReferences. We also need to explicitly capture the edges such as containment to include in the comparison. Therefore, we have extended the original feature hierarchy in SAMOS (see Fig. 4) with (1) *AttributedNode*, which holds all the information of a vertex including domain-specific type, name, type, attributes as key-value pairs; and (2) *SimpleType*, which indicates whether an edge is of type *containment* or *supertype* (meaning superclasses as named in EMF). All the mentioned feature types belong to the *SimpleFeature* class and represent non-composite stand-alone features.

There are several implementation details worth mentioning about the extraction:

- Our current implementation covers almost the full Ecore meta-model for extraction, except EAnnotations, EFactories, and generic types. We also ignore a few attributes such as namespace URIs and prefixes. The reason for exclusion for all the elements mentioned

⁷ <http://www.win.tue.nl/~obabur/publications/JVLC18/>.

Fig. 2. Example dataset with four metamodels.

Table 2

Term incidence matrix for the four metamodels.

mm	FSM	State	Machine	Source	label	Initial	Chn.	...
M1	0.35	0.15	0.15	0.09	0.05	0.15	0	...
M2	0	0.15	0.15	0.09	0.05	0.15	0	...
M3	0	0.15	0.15	0.09	0.04	0.15	0	...
M4	0	0.15	0.15	0	0.04	0.15	0.18	...

Table 3

Pairwise distance matrix.

	M1	M2	M3
M2	0.61		
M3	0.56	0.10	
M4	0.72	0.81	0.79

was our initial assessment of those (based on anecdotal evidence of our preliminary studies and discussions with language engineers) as mostly insignificant model content.

- Observing that for most attributes in Ecore metamodels, a mode or

Fig. 3. Dendrogram of the examples. (For interpretation of the references to colour in this figure, the reader is referred to the web version of this article.)

default value exists, we encode only non-default values in our AttributedNodes to reduce the data size and speed up the comparison. The fixed values can be deduced either from Ecore metamodel itself (default value), or by inspecting the majority of cases in our dataset (mode value).

Fig. 4. Feature hierarchy in SAMOS: simple features for representing vertices and edges, and aggregate features for encoding structure.

- We push the type (eType) information (e.g. what EDataType is assigned to an EAttribute) into the AttributedNode itself, rather than representing it as a separate vertex connected to that node, which would be expected from a naive/plain graph-based extraction. Examples of the modified extraction can be seen in the vertices v_2 and v_6 in the upcoming example. It avoids the problem we observed with the settings of [33], in which our approach matched too many irrelevant features just because of matching types, for instance all EAttributes with the type EString.
- The framework can also be configured to do a normalizing pass on element names to convert them into a standardized corresponding lower and snake case, tokenized and lemmatized form.

To exemplify, see the graph in Fig. 5, where the vertices are displayed as domain type and name information with the rest of the attributes being hidden. The AttributedNode features to be extracted in

Fig. 5. A simplified graph representation for (part of) a model.

our approach are the v_0 to v_6 .

- $v_0 = \{name : BIBTEX, type : EPackage\}$,
- $v_1 = \{name : LocatedElement, type : EClass, abstract : true\}$,
- $v_2 = \{name : location, type : EAttribute, eType : EString, lowerBound : 1\}$,
- $v_3 = \{name : Bibtex, type : EClass\}$,
- $v_4 = \{name : entries, type : EReference, eType : Entry, unique : true, \dots\}$,
- $v_5 = \{name : Entry, type : EClass, abstract : true\}$,
- $v_6 = \{name : key, type : EAttribute, eType : EInteger, lowerBound : 1, upperBound : 1\}$.

Note that there are no features corresponding to the basic EDataType leaf nodes in the figure; these are folded into their parent as discussed above. We have seen how we can extract information from (meta)models. In the next section we use the extracted information to encode the structure of metamodels in order to perform clone detection via cluster analysis.

5.4. Encoding structure in n-grams and subtrees

Our extended version of SAMOS, supporting subtree extraction in addition to the previously existing options [31], has the following three feature settings:

- Ignore the model structure completely, use nodes as is: i.e. the *unigram* setting [27],
- Encode structure in linear chunks: i.e. the *n-gram* (with $n > 1$) setting [31],
- Encode structure in fixed depth subtrees: i.e. 1-depth setting to be described in this paper.

In terms of the conceptual feature hierarchy in Fig. 4, one can think of unigrams as corresponding to SimpleFeatures, while *n*-grams and subtrees (potentially of depth $n > 1$) are aggregated features containing multiple SimpleFeatures. Based on the graph representation of a model, we can describe *n*-grams as *n* consecutively connected vertices. In contrast to [31] which simply omitted the edge information, we also incorporate the edges in the *n*-grams as SimpleTypes in order to represent the edge information as well. Although it is quite intuitive, the

Fig. 6. Example 1-depth subtrees from Fig. 5 and vertices above.

readers are referred to [31] for more details on the graph traversal and n -gram extraction. Some bigrams ($n = 2$) from Fig. 2 would be:

- $b_1 = (v_0, \text{contains}, v_1)$,
- $b_2 = (v_1, \text{contains}, v_2)$,
- $b_3 = (v_5, \text{hasSupertype}, v_1)$.

For this paper, we have extended the structure encoding with another option, namely subtrees. Our motivation was twofold: some shortcomings of n -gram encoding in terms of accuracy (please refer to our first case study in Section 6.1 and our previous work [14] for details), and the fact pointed out by Störle that UML class diagrams are lightly connected trees [8]. We expect this to be the case for metamodels as well, though an empirical assessment remains as future work. Currently SAMOS has been extended for subtree extraction with depth of 1 only (unlike arbitrary n -gram extraction for any n), and hence we show here only 1-depth subtrees. In principle, this can be extended to subtrees of arbitrary depth $n > 1$ as well, but is left as future work. Similar to the n -grams, based on the underlying graph for a model, we can describe 1-depth subtrees as a root node with all its child vertices. Some 1-depth subtrees from the example are shown in Fig. 6.

The subtrees are extracted in a relatively intuitive form as shown above. Note that the edges are pushed before the vertices (e.g. *(contains, v₁)*), effectively representing each such pair as an n -gram in a single tree node. This is an implementation choice, made due to our wish to use an existing tree edit distance library as a black-box, to be introduced in in Section 5.6. A naive extraction with edges in separate tree nodes would lead to a default similarity between most subtrees as they naturally have many relations in common—for example, many model elements *contain* other elements. With the containment tree nodes matched by the algorithm, too many subtrees would yield high similarities (for instance, close to 50% similarity merely due to matching edges). We avoid this by collapsing the original tree structure to obtain trees as represented in Fig. 6. In this way, we can control the n -gram comparison and make sure those with only matching edges do not yield a high similarity value.

5.5. Vertex and n -gram comparison

As aggregate features consist of multiple Feature vertices, we first define the extended vertex comparison to account for attributes with the following multiplicative *vertex similarity* formula:

$$vSim(n_1, n_2) = nameSim(n_1, n_2) * typeSim(n_1, n_2) * eTypeSim(n_1, n_2) * attrSim(n_1, n_2) \quad (1)$$

$$attrSim(n_1, n_2) = 1 - \frac{\#unmatched \text{ attributes between } n_1 \text{ and } n_2}{total\#attr. \text{ for that domain type}} \quad (2)$$

where *nameSim* is the NLP-based similarity between the names; while *typeSim* and *eTypeSim* are between the domain types and eTypes in the vertices, respectively. The last part *attrSim* measures the similarity for the remaining attributes.

As an example, the number of *unmatched attributes* in Eq. (2) would equal 1 given a variant of v_4 , see Fig. 5, with *unique: false* instead of *unique: true*. Note that the simple multiplicative formula in Eq. (1) might not be optimal, and has issues with certain situations; for

example, zero matching attributes leading to zero similarity. We leave it as future work to develop a more suitable scheme. The framework allows relaxing the similarity multipliers: e.g. by instead using a reducing multiplier of 0.5 for non-matching types, ignoring attributes altogether, and so on. N -gram comparison is the semi-relaxed formula [31]; i.e. given n -grams $\vec{v}_1$ and $\vec{v}_2$ with $2n - 1$ elements (n vertices and $n - 1$ edges corresponding to $v_1^{1..2n-1}$ and $v_2^{1..2n-1}$), the n -gram similarity is:

$$nSim(\vec{v}_1, \vec{v}_2) = ctxMult(\vec{v}_1, \vec{v}_2) * \sum_{i=1}^{2n-1} vSim(v_1^i, v_2^i) \quad (3)$$

$$ctxMult(\vec{v}_1, \vec{v}_2) = \frac{1 + \text{Inonzero } vSim \text{ matches between } \vec{v}_1 \text{ and } \vec{v}_2}{1 + (2n - 1)} \quad (4)$$

As an implementation workaround, we perform a corrective pass to eliminate *edge-only* matches. For example, SAMOS ignores matches such as A-contains-B vs. C-contains-D when A-C and B-D are completely dissimilar (i.e. $vSim(A, C) = 0$, $vSim(B, D) = 0$); effectively discarding the similarity due to matching merely *contains*. Finally, regarding the NLP part used to compute *nameSim*, SAMOS supports:

- normalizing in cases of mixed-casing and use of other conventions such as camel and snake case,
- tokenization and compound word similarity,
- stemming and lemmatization,
- Levenshtein distance for typos,
- WordNet-based measure of synonymy and semantic relatedness.

The vertex and n -gram comparison, as given above, provide a baseline for comparing more complex structures such as trees. Next we explain two techniques for subtree comparison.

5.6. Subtree comparison with ordered tree edit distance

For comparing (meta)model subtrees (of any depth) we have integrated the APTED⁸ ordered tree edit distance library [34] into SAMOS. APTED is a state-of-the-art algorithm for computing ordered tree edit distance in a memory efficient manner, using an optimal all-path strategy. As mentioned above, we would like to mostly use the algorithm as a black-box with two minor (plug-and-play) customizations: node data (to contain the n -grams as tree nodes), and the edit distance cost model. The latter is especially important for assigning the cost of minor changes in a non-binary manner; for instance, just changing the abstract modifier of an EClass would lead to a 5% change only. Next we elaborate those two important customization steps for our adoption of APTED.

Data representation. We extend the data structure to hold the necessary information in the APTED nodes. Consistently with the subtree examples of Fig. 6, a node can hold a simple feature (i.e. root) or a bigram (i.e. edge + another simple feature). In both cases each APTED node contains an n -gram.

Cost model. APTED uses a cost model to customize different weights (costs) for tree edit operations. We specify the following for the cost of

⁸ <http://tree-edit-distance.dbresearch.uni-salzburg.at/>.

node remove, insert and rename operations used by the algorithm:

- Deleting and insertion with a fixed cost of 1.0,
- Changing (i.e. renaming in the terminology of APTED) a node holding the n -gram p_1 into p_2 with the cost $1 - nSim(p_1, p_2)$. That is, ranging from 0 (exactly same), to 1 (completely distinct). Any value between captures the situation for changing several parts of the node types, attributes and so on.

Pre-sorting the tree nodes. As APTED works with ordered trees, we first sort the children of each node with respect to their types, names and attributes. We use a lightweight sorting algorithm at each depth level, hence only consider information at that level. The motivation comes from the fact that in metamodels the order of the child nodes do not matter, and the approach is consistent with the sort-and-compare approach in clone detectors such as NICAD-SIMONE.

Subtree similarity. Given two pre-sorted subtrees t_1, t_2 and $size$ being the number of nodes in the subtree, we define the edit distance based tree similarity in SAMOS as:

$$tSim_{APTED}(t_1, t_2) = 1 - \frac{APTED(t_1, t_2)}{\max(size(t_1), size(t_2))} \quad (5)$$

where $APTED$ is the tree edit distance between the two input trees, and $size$ yields the number of leaves on a given tree.

Note that we have left the integration of an unordered tree edit distance algorithm (such as Zhang–Shasha [35]) as future work, as a more powerful but costly alternative – especially for subtrees with high depth. Two major reasons would be the lack of an available implementation and the computational complexity. Instead we have developed the (unordered) comparison method for 1-depth subtrees only, as presented in Section 5.7. Note that the advantage of using a generic tree edit distance algorithm such as APTED is the uniform applicability for any tree with arbitrary depth.

5.7. Subtree comparison with modified Hungarian algorithm

For comparing unordered trees of 1-depth, we employ the Hungarian algorithm [36] for the child nodes. Given 1-depth subtrees t_1, t_2 , with parent nodes p_1, p_2 and their respective child node sets C_1 and C_2 , we define the Hungarian algorithm based tree similarity in SAMOS as follows. Note the distance formulation as reverse similarity, where $vDist(c_1, c_2) = 1 - vSim(c_1, c_2)$.

$$tSim_{HUNG}(t_1, t_2) = 1 - \frac{vDist(p_1, p_2) + \text{hungarian}'(C_1, C_2)}{1 + \max(m, n) + (\# \text{ matches in } C_1, C_2 \text{ with } vDist = 1)} \quad (6)$$

where we define the $\text{hungarian}'$ distance, slightly modified over the classical Hungarian distance as:

$$\text{hungarian}'(C_1, C_2) = \argmin_x \sum_{c_1 \in C_1} \sum_{c_2 \in C_2} \text{cost}'(c_1, c_2) x(c_1, c_2) \quad (7)$$

with slightly modified cost function cost' and match function x (as well as additional restrictions) as:

$$\text{cost}'(c_1, c_2) = \begin{cases} vDist(c_1, c_2) & \text{if } vDist(c_1, c_2) < 1 \\ 2 & \text{otherwise} \end{cases} \quad (8)$$

$$x(c_1, c_2) = \begin{cases} 1 & \text{if } c_1 \text{ and } c_2 \text{ are matched} \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

Given two sets (of possibly unequal sizes), we impose a further restriction to ensure that elements of the smaller set are matched exactly once; while some elements of the larger set can remain unmatched. Furthermore, in the second part of the cost function in Eq. (8), note the cost of 2 when the two n -grams are completely distinct. We try to mimic the cost of first removing the node and adding a completely new one:

represented as two operations instead of a single one (i.e. substitution). We try to find the optimal 1–1 match between all the child nodes of the trees using the vertex similarities for all pairs.

5.8. Weighting

SAMOS supports type-based weighting for features. So we can exploit type-based weighting to reduce the relative importance of certain features. Features with arguably more important types (based on our experience) are given higher weights; for instance, matching EClasses should be favored over e.g. matching EParameters. As a larger example, a 1-depth subtree with an EClass node at the root can be allocated a higher weight than a 1-depth subtree with an EOperation root. Note that these weights can be and do get configured according to the model types and problem statement at hand.

For simple types, we use a slight variation of the settings in [27], where we basically give higher weight, for instance, to EClasses, EPackages with lower weight for EReferences, EAttributes and finally even lower for EParameters and EDataTypes. For containment edges in aggregate features, we simply take the average of the type-based weight of all the vertices contained in that feature. For other edges which we consider less significant such as supertypes, we also assign a lower weight. The exact weights are omitted for space constraints; please see the supplemental material⁹ for the full list of type weights applied in this paper.

5.9. VSM calculation

SAMOS offers two *modes* for calculating the VSM: linear and quadratic (all-pairs). Linear VSM counts the exact occurrences, that is, computes the total frequencies of features with a single-pass, hence having linear complexity and much faster feature comparison (strict equality exploiting simple hashing). However, this mode is not able to account for synonyms or fine-grained differences in the features such as attributes or types. Quadratic VSM, on the other hand, compares each feature occurrence in the models with the entire feature set of the models, i.e. an all-pairs comparison, with expensive feature comparison such as tree edit distance, leading to a much more expensive (quadratically complex) computation but with proper treatment of fine-grained differences.

5.10. Distance measurement

Originally SAMOS adopted a regular VSM approach: a choice of a distance measure such as cosine and Manhattan, and calculation of the distance over the whole VSM. We have identified several shortcomings in the context of clone detection, and extended the distance measurement. We can summarize our arguments as follows:

- For clone detection, a measure is needed that is normalized (like cosine, and unlike Manhattan distance), yet size-sensitive (unlike cosine). There are several measures in the literature (and implemented in R) that fulfill the requirements of normalization and size-sensitivity, such as Bray–Curtis and Canberra [37].
- Bray–Curtis and Canberra may differ in terms of robustness in certain situations (depending on the dataset, outliers and distribution), but one important distinguishing feature is that Bray–Curtis is weight-sensitive, while Canberra is weight-agnostic. As a result, our weighting scheme, with e.g. higher weights to more important types of model elements, can be realized using Bray–Curtis distance, but not with Canberra.
- VSM assumes orthogonality of the features, and takes all columns into account for distance calculation. This is violated in our case

⁹ <http://www.win.tue.nl/~obabur/publications/JVLC18/>.

where our features are often (partly) similar to many other features and hence not orthogonal. For clone detection, we limit the distance calculation to the union of *originally contained* features by the two entities, rather than all the features in the dataset. Note that this solves only part of the problem; please see the case studies in Sections 6 and 7 for further discussion.

For the reasons above, we have integrated a masked variant of Bray–Curtis distance (Eq. (10)) into SAMOS, extending the default distance function in the R package *vegan*. Given an N dimensional vector space, data points P and Q for Ecore metamodels representing the whole model, an EPackage or an EClass depending on the scope; P consisting of features $P_1, \dots, P_m$ and Q consisting of features $Q_1, \dots, Q_n$, p and q the corresponding vectors on the full vector space for P and Q , the masked Bray–Curtis distance is over the vector subspace $P \cup Q$ (size $\leq m + n$) as:

$$\text{bray}'(P, Q) = \frac{\sum_i^{P \cup Q} |p_i - q_i|}{\sum_i^{P \cup Q} (p_i + q_i)}, \quad (10)$$

5.11. Clustering

As the final step of the SAMOS workflow, we apply clustering on top of the calculated distances to obtain the clone clusters. This in turn can be boiled down to finding non-singleton (size ≥ 2) and sizeable (size $\geq n$) groups of data points that are similar (distance $\leq t$); with n and t thresholds depending on the application scenario. While SAMOS originally supported k -means and hierarchical clustering, we have added and used *Density-Based Spatial Clustering of Applications with Noise* (dbscan) [38]. The algorithm (implemented in R package *dbscan*) uses two parameters, *minimum Points* (i.e. n) and ϵ distance (i.e. t) to compute density-reachable regions as clusters, with non-reachable regions being labelled as noise. While technical details are beyond the scope of this paper, dbscan possesses some properties that we desire for clone detection, namely:

- detection of clusters in various (non-convex, non-spherical) shapes,
- detecting noise, i.e. non-clones,
- suitability and efficiency for large datasets.

This concludes the details about extension we have integrated into SAMOS for this work. This extended version of SAMOS will be used in the case studies in the following section.

6. Case studies and comparative evaluation

We performed three case studies to evaluate the clone detection capabilities of SAMOS compared to NICAD-Ecore and MACH; in terms of accuracy, and with respect to scalability in the presence of thousands of models:

- *Case Study 1:* We analyzed artificially generated mutation cases and change scenarios where we measured pairwise distances with the base metamodel and the mutated ones to see how sensitive and accurate the different settings of SAMOS are, compared to NICAD-Ecore and MACH.
- *Case Study 2:* At EClass scope, we ran SAMOS with the most accurate setting along with NICAD-Ecore on the configuration management metamodels from ATL Zoo, and comparatively evaluated the clone pairs (and their correct classification, respective precision and recall) found by the two tools.
- *Case Study 3:* We performed a large-scale clone detection exercise, in two-steps with the cheapest and most expensive settings of SAMOS. We aimed to find the metamodel clone clusters in GitHub, for data preprocessing/filtering purposes and future empirical studies on metamodels and DSLs.

For the case studies, we used SAMOS version 0.5, NICAD 3.5 with our Ecore extension 1.1, and finally MACH 0.94. We have put all the relevant material (data, reports, validation/comparison and other information) as supplemental material on our website (see Footnote 9).

6.1. Case Study 1: mutation and scenario analysis

We have based this case study on a conceptual framework by Stephan et al. for validating our approach [39,40]. The framework suggests using mutation analysis to evaluate model clone detection techniques. We additionally investigated a few additional scenarios, which can be either conceived as chains of mutations or change scenarios (slightly similar to the ones in Roy [24]). In this section, we detail our assumptions, case design and goals. We finally present the results of SAMOS with different settings, MACH, and NICAD-Ecore; and discuss their potential strengths and weaknesses.

6.1.1. Case design

First we make the simplifying assumption that the scope for this case study is that of EClass clones. Inspecting the Ecore meta-metamodel, we identified a set of mutations, representing noteworthy small changes in an EClass as a starting point. Additionally, we designed some NLP-based cases, a few slightly more complex mutations involving move and swap operations, and finally some more complex scenarios to provide further insight into each tool's shortcomings. We outline our mutation sets as follows.

Set 1a. The first set of cases include 31 metamodels with atomic mutations. The mutations are mostly trivial operations of adding, removing or changing elements.

Set 1b. We added three specific mutations involving subtle element name changes: cosmetic renaming (such as lower vs. upper case, camel vs. snake case) and adding a typo, and replacing a word with its synonym, so that we can evaluate the NLP-related capabilities of the tools.

Set 2. For the second set, we have 5 mutations of reordering, moving and swapping model elements. The regular move mutation involves simply moving a model element elsewhere. In contrast, the extra move mutation (*moveSimilarContainer*) involves moving a model element A contained in B into a distinct container B' , with the condition that $vSim(B, B') = 1$ (same vertex similarity)—a change easily detectable using graph-based techniques but possibly not by approximate techniques such as ours. Swap mutations are designed in a similar manner. With this set, we wish to demonstrate a potential shortcoming of SAMOS (in some settings, due to the approximate nature of information representation) and other tools, even if (arguably) some of the cases might not be realistic in real data.

Set 3. The final set of cases includes 4 metamodels obtained from 3 base metamodels by one or more changes: (1) simple renaming of base metamodel elements which lead to a different lexicographic order (as sorting is an essential step of NICAD-* model clone detection and our ordered tree edit distance algorithm), (2) changing the eTypes of all EAttributes in the base metamodel, and (3) removal of one out of many occurrences of model element A contained in E (*removeNonOrthogonal*), versus (4) removing another model element which has no similarity to the rest of the elements in E (*removeOrthogonal*). Some of these may be trivial cases for e.g. a graph-based comparison, but possibly not for tools such as ours, MACH, and NICAD-Ecore.

We used medium-sized base metamodels to be manually mutated as indicated above. To give an impression, the base metamodel for simple mutations contained a single EClass with 5 EAttributes, 2 EReferences and an EOperation to be used for extraction and comparison at the EClass scope. We used the original files for SAMOS and their Emfatic transformations for NICAD-Ecore. For the UML part, we manually replicated all the mutations on UML class diagram representations of the same metamodels in order to evaluate MACH.

Table 4

Pairwise relative distances (reverse similarity) for SAMOS and NICAD-Ecore, absolute similarities for MACH compared to the identity similarity. Values in bold indicate the most problematic cases.

Set	Id	Mutation	Unigram	Bigram	tr-edit	tr-hung	NICAD	MACH
	1	addEClassSupertype	0.000	0.042	0.063	0.026	0.100	245/510
	2	addEClassEAttribute	0.025	0.049	0.030	0.025	0.100	245/510
	3	addEClassEOperation	0.025	0.051	0.145	0.123	0.100	129.5/510
	4	addEOperationEException	0.000	0.010	0.094	0.061	0.100	510/510
	...	...	...	...	...	...	...	...
	5	changeEClassNameRandom	0.114	0.737	0.033	0.027	0.100	NA
	6	changeEClassSupertype	0.000	0.011	0.055	0.067	0.100	510/510
	7	changeEClassAbstract	0.026	0.062	0.008	0.007	0.100	476/510
	...	...	...	...	...	...	...	...
	8	changeEAttrNameRandom	0.054	0.005	0.033	0.049	0.100	442/510
	9	changeEAttrType	0.040	0.003	0.012	0.010	0.100	510/510
	...	...	...	...	...	...	...	...
	10	removeEClassSupertype	0.000	0.048	0.038	0.042	0.100	229.2/510
	11	removeEClassEAttribute	0.028	0.054	0.034	0.058	0.100	148.1/510
	12	removeEClassEOperation	0.022	0.071	0.227	0.200	0.100	107.6/510
	...	...	...	...	...	...	...	...
	13	changeEAttrNameCosmetic	0.000	0.000	0.000	0.000	0.100	442/510
	14	changeEAttrNameTypo	0.003	0.001	0.001	0.001	0.100	442/510
	15	changeEAttrNameSynonym	0.001	0.001	0.068	0.001	0.100	442/510
	...	...	...	...	...	...	...	...
	16	changeOrder	0.000	0.000	0.000	0.000	0.000	1196/1196
	17	moveSimple	0.000	0.001	0.047	0.019	0.060	1196/1196
	18	moveSimilarContainer	0.000	0.000	0.046	0.018	0.060	1196/1196
	19	swapSimple	0.000	0.000	0.000	0.007	0.060	1196/1196
	20	swapSimilarContainer	0.000	0.000	0.000	0.000	0.060	1196/1196
	21	renamingReordering	0.009	0.009	0.100	0.008	0.480	130/338
	22	fullRetyping	0.248	0.045	0.101	0.101	0.530	338/338
	...	...	...	...	...	...	...	...
	23	removeOrthogonal	0.026	0.032	0.030	0.030	0.100	187.2/1066
	24	removeNonOrthogonal	0.051	0.053	0.052	0.052	0.100	220.1/1066

6.1.2. Goals for the distance measures

With respect to accuracy, we set five requirements for distance measures to achieve with SAMOS, and which we advocate as requirements for model clone detection tools in general. We use these to qualitatively evaluate the accuracy of the tools.

- R1. Obtain zero distance for mutations leading to Type A clones.
- R2. Obtain a non-zero distance for mutations leading to higher level clones such as Type B and C.
- R3. For those positive distances in R2, have the distance reasonably small (e.g 0.05 or 0.10 as a breaking point towards Type C) given that all the mutations in this section are small.
- R4. For those positive distances in R2, have them matching an intuitive assessment of distance based on the significance or weight of the change, for instance, changing just an EParameter is less significant than changing an EClass.
- R5. Overall have *bigger* changes lead to higher distances than *smaller* ones do. For instance, introducing a typo in a name should lead to lower distance than a complete renaming; and changing a type to a lower distance than removing the corresponding element altogether.

In short, we desire accurate but also fine-tuned distances for small changes. This is so because we want to correctly detect cases with accumulated multiple small changes as clones too, such as a complete change of EAttributes types (see Set 3 in Table 4), or changing all model element names with snake case into camel case. Overly large distances for small changes, when accumulated across multiple instances, might in fact lead to an incorrect clone classification or missing the clone altogether. This is why we were not satisfied with Set 1 only for evaluation, but also included Set 3. We believe such accumulated changes are interesting and complex scenarios for clones, which have been overlooked by the literature for clone detection evaluation, especially with the mutation-based approaches.

6.1.3. Evaluation

We have applied our clone detection technique and report here the distance measure between each case from the three sets and the corresponding base metamodel/UML model. We have used SAMOS with different feature settings (unigrams, bigrams, 1-depth subtrees with ordered tree edit distance, 1-depth subtrees with Hungarian distance) using weighted masked Bray–Curtis distance on the one hand; NICAD-Ecore and MACH on the other hand. Table 4 gives a representative subset of the results; for the full set of results, please see the supplemental material¹⁰. The numbers for SAMOS and NICAD-Ecore represent normalized similarity scores of each case with respect to the base metamodel. Since MACH did not output a normalized distance measure but an absolute one, we reported it along with the identity similarity (base model versus itself) for reference. The most prominent errors are presented in bold in the table.

SAMOS evaluation. In general, the results look promising, though not without certain errors and weaknesses for particular settings. Note that some of the results deviate from our previous work in [14] due to a different application of NLP parameters and thresholds, and a different weighting scheme.

- R1 is not violated by any technique in SAMOS in the two cases with cosmetic changes (mutation 13) and reordering (mutation 16).
- R2 is violated in a number of cases for SAMOS. Unigrams (not unexpectedly, as they ignore structural context) evaluate quite some mutations with zero distance, such as mutations 1, 6, 10 involving supertypes and 4 involving exceptions. Furthermore move/swap mutations (mutations 17–20 in Set 2) are mostly undetected by our technique. Even with the tree settings, SAMOS fails to detect some swap operations: mutations 19 and even 20 for *tree-hung*.
- R3 is not uniformly satisfied but the majority of the results are acceptable given a relaxed threshold range of 0.05 – 0.10. For the

¹⁰ <http://www.win.tue.nl/~obabur/publications/JVLC18/>.

mutations involving EOperations (numbers 3 and 12, especially for the tree settings), SAMOS returns relatively large values. While we hope to improve this in the future versions of SAMOS, this is not too problematic considering that removing an EOperation effectively removes all its content as well (EParameters, EExceptions); hence leads to a large distance. Besides, one of the biggest issues, as put in bold in the table, is with bigrams for renaming an EClass (mutation 5): due to the nature of feature extraction (i.e. bigrams), vertices with high number of outgoing edges (such as EClasses typically having many elements) are *over-represented* in the vector space. They are present in many features, hence any change leads to a larger distance in the vector space.

- R4, thanks to the type-based weighting applied, is improved over the non-weighted setting for n-grams as in our previous work [14]: we obtain larger distances for more significant changes, for example, in EClass vs. EAttribute vs. EParameter. For the tree settings however, due to several reasons such as EOperations being represented as separate trees in the VSM, we sometimes end up with non-optimal distances (mutation 12).
- Finally, R5 is relatively well achieved overall. We also would like to highlight the special case with renaming-reordering (mutation 21), which *tr-edit* scores quite dissimilar to the original, due to the lexicographic re-ordering involved. The same problem applies to the case *changeEAttrNameSynonym* (mutation 15), where replacing an EAttribute's name with its synonym leads to a different ordering of the elements in the EClass. Furthermore, element orthogonality affects the pairwise distance in all of the settings (i.e. a fundamental issue with our VSM-based approach), evident in the bottom cases *removeOrthogonal-removeNonOrthogonal* (cases 23, 24).

NICAD-Ecore evaluation. NICAD-Ecore overall does a good job in not missing many of the changes. It violates R1 only for the cosmetic EAttribute change case (mutation 13), due to its lack of NLP capabilities. Other than that, it satisfies R1 perfectly with the move/swap cases. Overall it has no problem with R2 too. R3 for the basic cases is satisfied, however the scenarios *renaming-reordering* and *fullRetyping* (cases 21, 22) are not properly assessed by NICAD; respectively due to LCS and line-based granularity for comparison. This line-based (and by default unweighted) approach further leads to the violation of R4 and R5: most changes in this case study are interpreted as “one-line” change, leading to a uniform 0.10 distance; most problematic for the mutations 14 and 15 with very minor changes in element names. Note that although we have not emphasized these cases in the table as a big problem, they lead to quite some inaccuracy (evident in the next case study). Finally NICAD-Ecore is not sensitive to orthogonality at all.

MACH evaluation. Inspecting the respective column overall, MACH (with the default setting of the closed-source distributable) seems to be making a few simplifications along the way just like SAMOS with certain of its settings. R1 is only violated for the cosmetic renaming scenario (mutation 13). In the corresponding papers MQ_{lone} is reported to have NLP capabilities, so we suspect they might be turned off in MACH. As for R2, however, MACH seems to be somewhere between the unigram and bigram settings of SAMOS. It seems to ignore or not consistently recognize changes to the following model parts: thrown exceptions (mutation 4), types (of properties, parameters; mutations 9 and others not shown on the table), modifiers, such as the attribute *unique*, and cardinalities. Note that some of these are not shown on the table due to space limitations; please see the supplemental material for details. Since MACH returns an absolute measure of the distances, we choose not to discuss R3-5 (for instance, through a relative measure with respect to the identity comparison, which might lead to a misjudgment). However, we further report that MACH did not find the proper class clone when the class name is changed (*changeEClassNameRandom*, mutation 5) and fails to detect move/swap changes (Set 2). There is also the surprising finding that MACH

correctly detected the addition and removal of a supertype, while failing to detect the change thereof (mutation 6). As a final observation MACH, like SAMOS, is sensitive to orthogonality of the changed model elements.

6.1.4. Discussion

The mutation and scenario analysis allows us to shed light on the capabilities, simplifications and inaccuracies of the tools compared. Not surprisingly, SAMOS has an increasing accuracy with more complex features and comparison techniques, reaching the best accuracy with 1-depth subtrees and Hungarian distance, at the cost of lower performance and scalability. With NLP as a core capability of SAMOS by design, it handles textual changes better than the other tools. The orthogonality issue is something fundamental to VSM and hence SAMOS with its current design, yet it is surprisingly observed in MACH as well. In any case, this issue is an interesting finding not only in terms of evaluating MACH, but also for the clone detection research in general when evaluating other tools as well. We plan to investigate in the future whether this should be regarded as a positive or negative aspect in clone detection. Finally for SAMOS, we plan to implement specific and more advanced weighting schemes (especially for the tree settings), so that it better meets our goals. As for NICAD-Ecore, it catches all of the non-NLP-related changes, however we believe its line-based granularity and LCS poses a problem. Note that (some of) these might not be fundamental flaws of NICAD itself, but we have taken the standard mode of operation of NICAD-SIMONE and just applied it to metamodels. Yet we believe the problem cannot be properly solved simply by applying pre-/post-processing and normalization steps of the types, modifiers and so on. One might need to change the line-based approach into a more fine-grained comparison.

A final remark is that it is unclear to us whether and how NLP could be properly integrated into NICAD (due its use of pre-sorting and LCS), and MACH (due to its use of Prolog indexing and pattern matching). While normalization such as supported by NICAD may avoid pairwise comparison of words with difference casing or typos, it will certainly not be able to avoid pairwise comparisons for more complicated cases such as semantic relatedness of model element names, for instance, using WordNet.

6.2. Case Study 2: metamodels in ATL Zoo

With the results of the first case study giving an idea about the tools, we proceed to the second one where we compare SAMOS with its most accurate setting, namely *tree-hung*, with NICAD-Ecore on a real dataset. We chose NICAD-Ecore over MACH for its higher accuracy, and since MACH lacks several features such as clone clustering, clone classes, and normalized similarity. As the dataset, we chose the conference management metamodels from the ATL Zoo,¹¹ due to the visibility of the public ATL Zoo and the fact that the conference management metamodels form a coherent thematic subset; thus potentially include many clones.

Methodology. Given the 14 conference management metamodels from ATL Zoo, we performed the following steps for SAMOS and NICAD-Ecore:

1. Granularity: extract all EClass fragments along with their content (transitive closure of containment),
2. Filtering: remove EClasses without any content or deemed too small (e.g. number of lines < 5 for NICAD-Ecore),
3. Clone detection: On the extracted EClass fragments, run SAMOS with *tree-hung* setting as in case study Section 6.1 and NICAD-Ecore with the distance thresholds: 0, 0.1, 0.3 (respectively Type A, B, C clones),

¹¹ <http://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Ecore>.

4. Validation: inspect (manually, but using EMFCompare¹² where possible) random subsets as given below for Type A, B and C. Note down the differences and categorize manually what clone class they belong to. The validation involved the following sets:

- Clone pairs common for both, $\text{SAMOS} \cap \text{NICAD-Ecore}$,
- Clone pairs $\text{SAMOS} \setminus \text{NICAD-Ecore}$,
- Clone pairs $\text{NICAD-Ecore} \setminus \text{SAMOS}$.

6.2.1. Results

Table 5 depicts the relevant numbers in our findings. Note that the number of clone pairs for NICAD-Ecore are extracted from the clone clusters (i.e. taking each pair in all the clusters), and differ slightly from the original clone pairs reported by NICAD-Ecore, which are 591 for Type B, and 1054 for Type C (see supplemental material¹³ for the clone pair report of NICAD-Ecore). This is due to the fact that NICAD uses connected component analysis for building the clone clusters from pairs. SAMOS also adopts a similar approach with its clustering. A first impression of the table is that SAMOS claimed to detect strictly more clone pairs than NICAD-Ecore for all three clone types, while there was disagreement especially in the B–C categories.

As mentioned, we relied on manual validation to assess the accuracy of the tools. We inspected random subsets of the common and different pairs found by the two tools. For the latter, we focused on the exclusively new pairs only – as by definition a Type A clone pair is a Type B/C clone pair too. We performed a random sampling with confidence level 90% and margin of error 0.10 to get the final sets for manual validation. Table 6 gives the resulting set sizes. The validation sets and the manual annotations can be found in the supplemental material.¹⁴ Note that in higher level clones, relatively minor changes are generally not annotated, such as cosmetic changes in Type C clone annotations.

Type A clones. SAMOS did a better job in finding the Type A clones, as the dataset includes many cosmetic changes in the model element names and attributes: upper/lower case and extra underscore characters. While the common set for both tools is completely accurate, the difference set is fully in favor of SAMOS. We considered the occasional cases with reordering of elements trivial (and perfectly detected by the two tools), and did not annotate them in our report. We traced the ones missed by NICAD-Ecore in the higher level clone reports and found that NICAD-Ecore detects them with large distances varying from 8 to 73 (i.e. considering each cosmetic change as a one-line difference).

Type B clones. The commonly found Type B clones were all correct, containing missing EOpposite values and few additions/removals of elements and supertypes. The exclusively Type B clones found by NICAD-Ecore (i.e. Type B_{ex}(NICAD)), in turn, entirely consists of actually Type A clones misclassified by NICAD-Ecore. Type B_{ex}(SAMOS) has several categories of clone pairs. In the simplest category, there are pairs with very small difference (missing EOpposite, type changed from String to Integer, or interestingly a metaclass change while keeping the element name the same), which are clearly Type B – so correct classification by SAMOS. In other cases there are multiple changes in large EClasses; there, weighting and orthogonality of the changes cause SAMOS to classify those as Type B clone pairs. While in most of these cases the pairs are highly similar to each other, it is difficult to strictly label them as Type B, C or higher. We would like to exemplify a few of those: with a base EClass with elements such as *hasMail*, *writePaper*, *submitPaper*, newly added elements such as *hasEmail*, *submit*, *writeArticle* would all be considered highly similar to their base counterparts due to SAMOS' NLP functionality such as typo checking, compound word similarity and WordNet semantic relatedness. Those cause SAMOS to calculate a higher similarity score compared to the addition of orthogonal elements.

Table 5

Number of clone clusters and pairs found in the conference management metamodels. *c* denoting the number of clusters, *p* pairs and *ex* exclusively new pairs (i.e. Type B–Type A, and Type C–Type B).

	Type A		Type B			Type C		
	#c	#p	#c	#p	#ex	#c	#p	#ex
SAMOS	62	702	59	1142	440	40	1687	546
NICAD	58	523	70	599	66	52	1264	665
SAMOS \ NICAD		179		543	405		426	381
NICAD \ SAMOS		0		0	41		0	498
SAMOS $\cap$ NICAD		523		599	35		1264	167

Table 6

Sizes of the validation sets.

	Type A	Type B _{ex}	Type C _{ex}
SAMOS \ NICAD	50	58	58
NICAD \ SAMOS	–	26	60
SAMOS $\cap$ NICAD	60	24	49

While this may be considered controversial—and we have therefore conservatively labeled them as Type C, i.e. misclassified by SAMOS—we believe this to be a powerful feature of our framework.

Type C clones. The changes observed in this category mostly involved multiple addition/removal/changes of model elements, along with above mentioned changes such as cosmetic changes, changes in attribute values (not documented in our annotations). The intersection of the two tools contains mostly correctly classified clone pairs of such type, except a few cases where additional minor changes, such as a new supertype, led to a high percentage difference in small EClasses.

Type C clones exclusive to NICAD-Ecore are mostly misclassified Type A/B clones due to NICAD's line-based granularity and lack of NLP. Two examples are a single supertype addition and a cosmetic renaming, and a type change from String to Integer, both of which lead to NICAD-Ecore classifying those as Type C clones. In a few cases however, NICAD-Ecore does a better job while SAMOS underestimates those as Type B. For the opposite set (SAMOS \ NICAD), all detected pairs are interesting cases with considerable similarity. While most are due to the addition of (multiple) model elements, there were two interesting cases where all the EAttributes of an EClass were replaced with EReferences with the same name and type name (date of eType Date). Nevertheless many pairs we manually assessed have relatively too many changes (due to weighting, orthogonality) and hence cannot be considered Type C clones (denoted as $> C$ in our report in the supplemental material¹⁵). In such cases however, it may be that the elements of the pair are in fact part of a non-spherical (for instance stripe-like) cluster in higher dimensional space, and hence in fact are (indirectly) related. SAMOS performs *reachability clustering* (similar to the connected component analysis in NICAD) and can detect such non-spherical clusters. The implication is, it is not guaranteed that all the pairs in the cluster have low (i.e. lower than threshold) pairwise distances; in fact the ends of the stripe can have much higher distances than the threshold.

Although it is hard to exactly pinpoint the clone type in this set, we annotated the pairs being C, C borderline (arguably C) and $> C$. We report the accuracy assuming the worst cases for SAMOS, i.e. all borderline cases being incorrect. Table 7 gives a final account of both tools' precision and relative recall separately for all the validation sets. The number of real clones (as labelled manually) necessary for relative recall is calculated as the following: (1) find the percentage of the relevant clones in *all* the validation sets, (2) extrapolate the percentages to the original sets with respect to the validation sets (i.e. multiply

¹² <https://www.eclipse.org/emf/compare/>.

¹³ <http://www.win.tue.nl/~obabur/publications/JVLC18/>.

¹⁴ <http://www.win.tue.nl/~obabur/publications/JVLC18/>.

¹⁵ <http://www.win.tue.nl/~obabur/publications/JVLC18/>.

Table 7

Precision and relative recall of SAMOS and NICAD-Ecore aggregated from all the validation sets.

		Type A	Type B _{ex}	Type C _{ex}
SAMOS	Precision	1.00	0.89	0.65
	Rel. recall	1.00	0.92	0.78
NICAD-Ecore	Precision	1.00	0.46	0.26
	Rel. recall	0.75	0.09	0.34

percentages with the sizes) and (3) average it over all the sets. Note that step (1) means we exhaustively looked at Type B and C sets even when looking for Type A clone pairs, and so on. As for the extrapolation step, we rely on the underlying assumption that random selection of the considerable-sized validation samples ensures similar statistics (percentage of clones in particular) as the population, i.e. the full set of clones, which the sample was taken from.

6.2.2. Discussion

In this case study, we performed comparative clone detection on a real dataset. We found evidence to support our claims in case study 1: we need NLP and fine-tuned distances so that multiple small changes accumulate properly to a reasonable sum. NICAD-Ecore overestimates distances due to the lack of these features. On the other hand, SAMOS seems in some cases to underestimate the distance, thus leading to some amount of inaccuracy as well. The results of this case study provided us with insight and helped to improve SAMOS – for instance with better weighting schemes, to begin with. The issue with respect to the orthogonality issue are left as future work to examine in detail, based on interviews with the developers or discussions within the clone detection community. Nevertheless, overall SAMOS performs well with both high accuracy and recall.

Note that we used NICAD with the basic mode of operation as done in SIMONE (i.e. normalization and line-based comparison with LCS). We believe, however, that all of the problems mentioned cannot simply be solved by normalization steps, but rather need more fine-grained (i.e. not line-based) comparison, employing NLP as well. NICAD is nevertheless in principle extensible in the sense that LCS, for instance, can be replaced with other techniques such as set similarity and so on.

We have not reported exact run times here (given the small size of the data). SAMOS is much slower than NICAD: for this case study, the former processes the data in the order of minutes, the latter in just a few seconds. In our case, this level of a performance loss is an acceptable trade-off for the increased accuracy.

For fairness, we would like to emphasize that one of the major weaknesses of SAMOS, namely the move-swap type of changes, are not found in this case study. We leave it as future work to compare the two tools with higher scope and/or model types with typically longer chain-like structure (state charts, or Simulink models already supported by NICAD-SIMONE), which might reveal additional shortcomings of SAMOS. Another interesting line of comparison would involve structural clones (such as blind renamed clones used by SIMONE), which SAMOS does not support at the moment.

6.3. Case Study 3 – metamodel clones in GitHub

We performed a third case study to apply SAMOS on real data mined from GitHub. The goal here is again to assess the accuracy of our tool, but additionally to demonstrate its scalability to thousands of metamodels. This level of scalability is necessary for empirical studies on metamodels and related DSLs in GitHub (with tens of thousands of items) or industrial DSL/MDE ecosystem evolution studies (with hundreds to thousands of items, and tens of versions per item). Next we detail the mining process and the iterative clone detection we performed.

6.3.1. Mining ecore metamodels from GitHub

For mining GitHub, we did not perform a rigorous methodology as it is not the main focus of this paper. We rather leave it for the follow-up empirical studies to tackle, along the lines of the related work by Kalliamvakou et al. [41] and Hebig et al. [42]. We do not guarantee a full coverage of all the metamodels in GitHub; we are aware that our search mostly covers just the master branches and we are not able to search for files bigger than 384 KB.¹⁶

Using the web interface of GitHub, we searched for Ecore metamodel files containing the word *EClass*.¹⁷ We crawled all the resulting metamodels and applied two filtering passes, respectively removing (1) files not parsable by EMF or not having any content, and (2) exact duplicates, using MD5 hash comparison.

The search yielded 68,511 code results,¹⁸ of which 68,485 could be found and downloaded. Filtering pass 1 removed 1337 files. Roughly 2/3 (45,355) were eliminated after exact duplicate removal, resulting in a final set of 21,793 metamodels. The high number of duplicates contrasts with the results on UML models in GitHub by Hebig et al., who reported merely 12% duplicates. While at first glance one can see that there are public datasets which duplicate the mined metamodels by publishing them again on GitHub, a detailed analysis and explanation of this fact is left for future studies. The clusters of exact duplicates can be found in the supplemental material.¹⁹

6.3.2. Iterative process for clone detection: methodology

The ~ 22k metamodels still posed a challenge for our framework with computationally expensive settings such as *tree-hung*. While we have put some effort into a distributed (big data) processing back-end for SAMOS [44], it is in a rather early experimental phase. Here we instead aimed to develop an iterative approach and do the assessment. Given various settings of SAMOS, the idea is to start with cheaper, less accurate settings to find clusters with the most promising clone candidates, and process each cluster separately with the more expensive and accurate settings:

1. Run SAMOS with full NLP on unigrams (no structure) of names only (no types, attributes) with threshold 0.3 and the rest of the settings as in the above case studies. However, pre-tokenize and pre-lemmatize all the names to reduce the vector space.
2. Remove the outliers (non-clones), get each clone cluster into a separate bucket.
3. For each bucket, run SAMOS separately with the *tree-hung* setting with attributed 1-depth subtrees (i.e. as in case study 2).
4. Find Type A (i.e. almost the same) and Type C clones (highly similar) with thresholds 0 and 0.3. We choose to omit Type B for simplicity, and the fact that Type C clones already cover a superset of Type B clones.
5. For validating the precision, inspect manually random subsets of the clusters (confidence level 90% and margin of error 0.10) as well as ones with large number of elements, > 10 for Type A and > 50 for Type C clones—as cases of higher interest.
6. For validating the relative recall of this new iterative approach (compared to running SAMOS on the whole data at once), run SAMOS with *tree-hung* random subsets assembled from the disjoint buckets and the whole *outlier bucket* to see if we can find clones which would not be found by the iterative approach.

Note that the preprocessed unigram setting in step (1) greatly reduced the vector space, allowing us to handle the whole data set at

¹⁶ <https://help.github.com/articles/searching-code/>.

¹⁷ Using the following URL: <https://github.com/search?q=EClass+extension%3Aecore>, same as what has been done by Kolovos et al. [43].

¹⁸ Performed on June 1st, 2018.

¹⁹ <http://www.win.tue.nl/~obabur/publications/JVLC18/>.

once. Even with the outlier removal at step (2), we ended up with too many metamodels. We rather processed each bucket separately; hence helping tremendously in dealing with the inherently quadratic complexity of our approach. Steps (5) and (6) allowed us to reach qualitative conclusions about the accuracy of SAMOS, and this iterative process. For step (1), we did not go for other approaches such as word stemming (heavier normalization), character n -grams or hash values (such as locality sensitive hashing [45]), in order to be able to use full NLP including WordNet semantic relatedness. An advantage with those approaches, though, is that we could calculate the VSM in a linear way (see Section 5.9 for the linear vs. quadratic computation of VSM). Even with the current unigram setting, running SAMOS on the dataset in this case study, but with complex parameters such as NLP turned off and using the linear VSM, only takes minutes (in contrast to hours when using the quadratic VSM).

6.3.3. Results

Step 1 of our process resulted in 3813 buckets involving 16,048 metamodels, with an average of 4.2 metamodels per bucket. This is yet another important empirical finding: again more than two thirds of the non-duplicate metamodels in GitHub are highly similar to others (provided that SAMOS detects them accurately; see discussion below). We found a few clusters with over 100 items, and one with over 2800. This might be due to the clustering algorithm (cf. discussion in case study 2 on spherical vs. non-spherical clusters), and will be elaborated further in the validation and discussion subsections.

Running *tree-hung* clone detection on the 3813 buckets resulted in 3680 Type A clone clusters containing a total of 8488 metamodels and 2200 exclusively Type C clusters involving 11,044 metamodels.

Type A clones. Inspecting the 67-item validation set for Type A, we were able to manually label all except one as Type A, hence correctly classified by SAMOS. We identified mostly whitespace differences, re-ordering of elements, cosmetic changes in names, XML encoding. Changes for instance in nsURI and nsPrefixes of EPackages, generic types and EAnnotations are by design ignored by SAMOS, so we did not consider those as incorrect. A number of cases involved differences in the eType object paths while the names remained the same, where they were treated by EMFCompare as identical as well. One cluster contained metamodels with unresolved proxies (hence eTypes not being retrieved by SAMOS and SAMOS conflating the resulting null values); SAMOS incorrectly labelled it as Type A despite the type differences. A surprisingly large percentage of the cases involved an interesting situation: probably copy-pasted metamodels in different platforms contain different new line characters, resulting in multiple additional *carriage return* characters. This not only happens in a few repositories, but seems like a GitHub-wide phenomenon. This simple fact renders simplistic file comparisons inadequate and might be taken into account by studies like ours (this paper and [11]) or the ones by Hebig et al. [42] and Kolovos et al. [43] (and replication studies thereof). We additionally checked the 11 clone clusters with more than 10 metamodels per cluster; 9 were correctly classified (except the parts ignored by SAMOS, notably EAnnotations – including any OCL constraints encoded in those): sliced UML metamodels, Petri nets, Ecore meta-metamodel, and various toy example metamodels. In the final two clusters, which we manually labelled as Type B, there were two minor problems: a small implementation error due to empty package not being represented in SAMOS, and token filter removing too short tokens. Note that the latter (together with stopword removal) is in most cases actually beneficial for ignoring insignificant changes; as an example consider the following set of *name_of_class*, *name_class*, *name_class_1*, *name_of_classX* and so on.

Type C clones. For validating exclusively Type C clones, we only considered whether the cluster contained *highly similar and interrelated* metamodels (adequate for our follow-up studies); rather omitting the exact categorization into Type B, C or higher. We inspected the 66-item

random validation subset plus 13 clusters with sizes larger than 50. The majority of the clusters contained highly similar metamodels, ranging from toy metamodels to ones for C++ , Java, fUML, petri nets, business process models. These were situated both intra and inter-repositories. We identified certain flaws of SAMOS as well: apart from a minor implementation error, this involved a case where orthogonality might also be an issue (as discussed in the previous case studies), and two cases where cluster connectivity led to multiple small and consistent sub-clusters getting packed into a large cluster. One of those involved too small metamodels with matching base classes (hence higher weight). In the second, we had a 677-item cluster with UML, SQL, state machine and graph metamodels all thrown in together (presumably because of reachability clustering).

Additional random runs. As a result of the 100 runs each with 100 metamodels randomly picked from the buckets and outlier set under the restriction that at most one metamodel per bucket was selected metamodels, we obtained zero Type A and zero Type B clones. However, we found 9 additional clone pairs (i.e. clusters with size two) of Type C. Of these, 4 were correctly classified, while one was borderline and 3 were incorrect. The inaccuracy stemmed presumably from a lot of matching tokens (XElement, YElement, ...) and/or (partly) matching supertypes which were ignored by the unigram pass. In the first case, using WordNet semantic relatedness measures led tokens such as Element and Component to be very similar (Lin similarity of 100%).

6.3.4. Discussion and performance

The results indicate that this iterative mode of SAMOS is able to detect many clones with high precision: 75/78 for Type A, and 76/79 for Type C clone clusters. The high precision is not so surprising given the above case studies used depth 1 subtrees with the Hungarian distance. While we cannot assess the absolute recall, we can claim that the iterative approach also has a very high recall with respect to the non-iterative mode of SAMOS: the 9 small clusters from the random runs are very few compared to the original 4073 clusters with > 10k models. Yet again a qualitative analysis for identifying the problematic cases (due to implementation error, or something fundamental with SAMOS) helps us to improve SAMOS in the future.

The VSM computation part and the distance calculation being the bottleneck, the complexity of SAMOS is essentially quadratic with respect to the number of features in the dataset. Expensive feature comparison techniques such as Hungarian algorithm and NLP (notably WordNet checks) also greatly increase the execution time. However, as mentioned above, the complexity can be lower when there are many common features across the dataset – reducing the vector space. Thanks to this, we were able to perform the first, unigram pass on a single core of an 2.3 GHz Intel Core i7 processor with 16 GB 1600 MHz DDR3 memory in approximately 14 h. The second pass, in turn, took close to 12 h in total for the 3813 buckets. Given the large size of the dataset, we believe SAMOS already achieves a good scalability. We plan to further improve the performance with optimizations and a distributed computing back-end, to cope with tens of thousands of models or even more.

We would like to emphasize that we leave the detailed empirical discussion of the results as future work. We believe it is worthwhile to investigate several phenomena, for instance, whether it is expected to have Type A clones for Ecore meta-metamodels as they are used across many EMF-based projects, and what are the syntactic, semantic and pragmatic differences across the various Petri Net metamodels. We also deliberately left out a comparative performance assessment of SAMOS vs. NICAD-Ecore. This is because we established the higher accuracy of SAMOS in the previous case study, and moved on to assess its scalability and feasibility for a large dataset. In future work, we plan to conduct case studies comparing the performance of SAMOS vs. NICAD-* and MACH as well.

7. Overall discussion and future work

In this section, we discuss several aspects of our approach and the techniques we developed.

Underlying framework. We have built our clone detection technique on top of SAMOS, exploiting its capabilities such as NLP and statistical algorithms for free. The framework easily allowed extension, for example with extraction of new features, and addition of new distance measures. This is one of the strengths of our approach, also considering recent developments within SAMOS such as support for distributed computing and more sophisticated NLP. Using R as the back-end enables us to further experiment with advanced statistical and data mining techniques. We are investigating whether to integrate SAMOS into a data mining framework which would enable us to utilize features such as a GUI, workflow management, and advanced visualization, data mining, and machine learning. We plan to publish SAMOS as an extendible open source framework for model clone detection.

Accuracy SAMOS has a high accuracy, as given in the three case studies. When compared to existing tools, it does a better job, notably thanks to its NLP capabilities (given our clone type categorization). While it is harder to give an account of recall, we tried to give some assessment of (relative) recall in case study 2 (vs. NICAD-Ecore) and 3 (vs. non-iterative execution). The qualitative analysis of the inaccurate cases for all three tools further allow us to pinpoint the weaknesses of the approaches and improve SAMOS where applicable in the future. We believe the variety of case studies and comparison with state-of-the-art clone detectors show SAMOS to be a powerful tool.

Performance and scalability. Overall, SAMOS is slower than NICAD-Ecore and MACH, which can mostly be attributed to the quadratic complexity of the all-pairs VSM and distance measurement, the complex NLP employed, and the expensive comparison algorithms performed. Nevertheless, SAMOS can handle quite large datasets (in the order of tens of thousands metamodels, millions of model elements) without too much of a problem, with the help of the iterative process introduced in case study 3, and the distributed computing back-end being developed.

Genericness. SAMOS is in principle generic, in the sense that it can be applied to any graph-based model provided one implements the corresponding (metamodel-driven or hard-coded) feature extraction. SAMOS has been or is currently being applied in different contexts for UML class diagrams, industrial DSLs, feature models [46] and state charts [47]. It will be further investigated to what extent our technique is applicable to these and other types of models, especially in data flow or block-based languages. Another challenge would be detecting structural clones e.g. for Simulink models, which would certainly require a considerable extension to SAMOS. We regard the detection of semantic clones more of a longer term goal.

Orthogonality and clustering. As discussed in all three case studies above, the orthogonality of changes among artifacts changes not only the assessment of (some of the) clone detectors, but also the assessment of the clone types especially along the borders of Type B, C and higher. It is not clear to us whether orthogonality plays a positive or negative role in the clone classification and perception. We leave it as future work to investigate this phenomenon in detail. Another important aspect is the connectivity clustering of spherical/convex vs. non-spherical/convex shapes. This is a well-known issue in the data mining domain, and we would like to investigate further how it affects the clone detection domain, and whether other clustering techniques might lead to more accurate groupings. For small models, for which even atomic changes lead to too high of a dissimilarity as we found out in our case studies, we plan to investigate solutions such as transforming the similarity function, or using some variant of the *binding strength* advocated by Störrle [12].

Improving SAMOS. With the input from the three case studies and our increasing insight working with different types of models, we plan to further extend and test SAMOS with additional features including

longer n -grams, arbitrary-depth trees, subgraphs, ordered elements, customized and improved weighting schemes (such as ways to remedy bigram weighting issue, or depth-based weighting), distance measures and statistical algorithms.

More powerful NLP. We believe NLP is crucial to tackle real world datasets, and is an underrated aspect in the current model clone detection literature. We plan to further improve the NLP capabilities of SAMOS with part-of-speech tagging, context-based word-sense disambiguation, and more advanced, possibly domain specific semantic resources i.e. beyond WordNet.

Other practical aspects. Several other practical aspects are reported in the literature as important for applying model clone detection in practice [7,39]. In this work, we have fixed scoping of EClass, EPackage and whole model, thus do not run into the nested clones problem [39]. This would be somewhat important for e.g. EPackage scope, but even more so for other types of models. Other aspects including clone ranking, reporting and inspection, visualization are also left as future work.

Threats to validity. A threat to validity of this study is the lack of an assessment of the absolute recall, although we used state-of-the-art (and hence presumably reasonably accurate) clone detector tools as comparison. Ways of further mitigating this problem would be to apply a proper and automated mutation-based assessment tool [48] on the one hand, but also perform further comparative studies with other clone detectors to have a stronger account of the overall relative recall of all the tools combined. The manual labelling of clones, hence the manual validation of accuracy, is also a labor-intensive and error-prone process; to reduce the error rate, we plan to incorporate multiple assessors with techniques from the empirical research domain, and work with domain experts from the industry and the clone detection community for building ground theory [49]. Other additional manual validation techniques could be employed, as proposed by Stephan et al. [50]. Moreover, further comparative case studies not only with NICAD and MACH, but also with additional model clone detectors, notably ConQAT, would lead to a more objective assessment of accuracy. We plan to publish SAMOS along with those validation data in the future for replication studies and reproducibility.

8. Related work

The bulk of model clone detection research follows and is inspired by the larger software clone detection [51] and code clone detection literature [5], which is not covered in this paper. The readers are referred to those extensive surveys for further information. Model clone detection research, which can be considered as a sub-area of model comparison [9] in a broader sense, is driven by the major approaches accompanied by tooling as outlined in Section 1: CloneDetective/ConQAT [4], SIMONE [10], MACH [8,12]. Other approaches based on SIMONE are presented by Anthony et al. [25] for UML sequence diagrams and Chen et al. [52] for MATLAB Stateflow models. These approaches use a lot of sophisticated language-specific and problem-specific pre-/postprocessing to extract meaningful model fragments (for instance, conversation patterns in the former) and rely on the mode of operation used in NICAD-SIMONE to find near-miss clones.

Graph-based approaches for MATLAB Simulink models include ConQAT [4] and ModelCD [11]. Both use graph matching and graph-theoretic algorithms to find clones in sets of data flow models. ConQAT is used in comparative studies such as the one by Stephan and Cordy [48] and is reported to have lower accuracy (especially recall) than NICAD for near-miss clones. ModelCD in turn, contains two algorithms, eScan and aScan, for clone detection, however they are not publicly available. Störrle reports a much higher performance and scalability for $M_{Q_{tone}}$ compared to ConQAT and ModelCD. eScan (for exact clone detection) is implemented by Strüder et al. [53] for detecting model transformation clones and compared against ConQAT. The authors report a much lower performance and scalability than ConQAT for eScan.

The performance issue with graph-based approaches is even more evident in more fundamental approaches, namely Similarity Flooding [26] or general (sub-)graph isomorphism.

The aScan approximate clone detection algorithm in ModelCD, however, follows a very similar approach with SAMOS in terms of model fragmentation and vector-based similarity calculation. While we started with IR-inspired scenarios for SAMOS for domain clustering of models, after moving to clone detection we realized that aScan follows a very similar approach to SAMOS. It extracts features for an approximation of the (sub)graphs: n -paths (similar to n -grams in SAMOS) and $(p-q)$ -nodes (similar to depth 1 subtrees in SAMOS). It then uses an occurrence-counting vector with a simple distance measure for calculating the similarity. Nevertheless, we can identify several distinguishing features: they (1) use a very simplistic representation of model information, for instance Simulink nodes by their types only, discarding all other information, (2) do not employ any fine-grained comparison such as NLP, and (3) use additional constraints and heuristics for computing the clones (especially in the case of nested clones). Unfortunately, given the unavailability of ModelCD and the lack of extensive comparative studies of ModelCD with other clone detector tools, it is impossible to assess ModelCD properly. Besides, like MACH, ModelCD is reported on a single model to find the clone within; SAMOS can operate on multiple models and finds clones across different models as well as within the models.

Another approach by Hummel et al. [54] tries to mitigate the shortcomings of fully-fledged graph-based approaches, by indexing subgraphs in terms of their canonical labels approximated by MD5 hashes. Those labels consist of tuples of relevant information such as name of the model files, lists and sequences of normalized statements in the subgraphs. They act as a heuristic step, after which a proper isomorphism check is performed on the potential clone pairs.

Another interesting approach involves clone detection for UML sequence diagrams [55]. The authors suggest flattening sequence diagrams into a 1-dimensional array to construct suffix trees, and identify longest common prefixes of those as clones. Ekanayake et al. present another graph edit distance-based clone detection approach, for business process models [56]. A final interesting reference is the DSL clone detection study by Tairas and Cabot [18], where the authors investigate Type I and II clones in Object Constraint Language. While the target language is a textual one, it is lifted to the model (abstract syntax) level and clone detection is done via model transformations. The authors note that their approach is not flexible enough to detect Type III (near miss) clones.

9. Conclusion

In this paper we have presented a novel model clone detection approach based on the SAMOS model analytics framework using information retrieval and machine learning techniques. We have extended SAMOS with additional scoping, feature extraction and comparison schemes, customized distance measures and clustering algorithms in the context of metamodel clone detection. We have evaluated our approach using a variety of case studies involving both synthetic and real data; and identified the strengths and weaknesses of our approach along with two other state-of-the-art clone detectors, namely NICAD-SIMONE and MACH. We conclude that SAMOS stands out with its higher accuracy while still being considerably scalable; it proves to be useful for further large-scale clone detection and empirical studies on metamodels and domain specific languages.

Acknowledgments

We would like to thank Jia Zhang and James Hay for contributing on clone detection with NICAD and SAMOS.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.cola.2018.12.002.

References

- [1] Ö. Babur, L. Cleophas, M. van den Brand, B. Tekinerdogan, M. Aksit, Models, more models, and then a lot more, in: M. Seidl, S. Zschaler (Eds.), *Software Technologies: Applications and Foundations*, Springer International Publishing, Cham, 2018, pp. 129–135.
- [2] J. Mengerink, A. Serebrenik, R.R.H. Schiffelers, M.G.J. van den Brand, Automated analyses of model-driven artifacts: obtaining insights into industrial application of MDE, *Proceedings of the IWSM-Mensura*, ACM, 2017, pp. 116–121.
- [3] R. Wester, J. Koster, The software behind Moore's law, *IEEE Softw.* 32 (2) (2015) 37–40.
- [4] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, S. Teuchert, Clone detection in automotive model-based development, *Proceedings of the Thirtieth International Conference on Software Engineering*, ACM, 2008, pp. 603–612.
- [5] C.K. Roy, J.R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: a qualitative approach, *Sci. Comput. Program.* 74 (7) (2009) 470–495.
- [6] C.J. Kasper, M.W. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software, *Empir. Softw. Eng.* 13 (6) (2008) 645.
- [7] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfahler, B. Schaezt, Model clone detection in practice, *Proceedings of the Fourteenth International Workshop on Software Clones*, ACM, 2010, pp. 57–64.
- [8] H. Störle, Towards clone detection in UML domain models, *Softw. Syst. Model.* 12 (2) (2013) 307–329.
- [9] M. Stephan, J.R. Cordy, A survey of model comparison approaches and applications. *Proceedings of the International Conference on Model-Driven Engineering and Software Development*, (2013), pp. 265–277.
- [10] M.H. Alalfi, J.R. Cordy, T.R. Dean, M. Stephan, A. Stevenson, Models are code too: Near-miss clone detection for Simulink models, *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2012, pp. 295–304.
- [11] N.H. Pham, H.A. Nguyen, T.T. Nguyen, J.M. Al-Kofahi, T.N. Nguyen, Complete and accurate clone detection in graph-based models, *Proceedings of the Thirty-first International Conference on Software Engineering*, IEEE Computer Society, 2009, pp. 276–286.
- [12] H. Störle, Effective and efficient model clone detection, *Software, Services, and Systems*, Springer, 2015, pp. 440–457.
- [13] Ö. Babur, Statistical analysis of large sets of models, *Proceedings of the Thirty-fifth IEEE/ACM International Conference on Automated Software Engineering*, (2016), pp. 888–891.
- [14] Ö. Babur, Clone detection for ecore metamodels using n-grams, *Proceedings of the Sixth International Conference on Model-Driven Engineering and Software Development*, 2018, (2018), pp. 411–419.
- [15] A.M. Sutti, M. van den Brand, T. Verhoeff, Exploration of modularity and reusability of domain-specific languages: an expression DSL in metamodel, *Comput. Lang. Syst. Struct.* 51 (2018) 48–70.
- [16] B. Combemale, J. Kienzie, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, A. Wortmann, Concern-oriented language development (COLD): fostering reuse in language engineering, *Comput. Lang. Syst. Struct.* 54 (2018) 139–155.
- [17] T. Kosar, S. Bohra, M. Mernik, Domain-specific languages: a systematic mapping study, *Inf. Softw. Technol.* 71 (2016) 77–91.
- [18] R. Tairas, J. Cabot, Cloning in DSLs: experiments with OCL, *Proceedings of the Forth International Conference on Software Language Engineering*, Springer, 2011, pp. 60–76.
- [19] L. Prechelt, G. Malpohl, M. Philippsen, Finding plagiarisms among a set of programs with JPlag, *J. UCS* 8 (11) (2002) 1016.
- [20] B. Al-Batran, B. Schätz, B. Hummel, Semantic clone detection for model-based development of embedded systems, *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, Springer, 2011, pp. 258–272.
- [21] J.R. Cordy, C.K. Roy, The NiCad clone detector, *Proceedings of the Nineteenth International Conference on Program Comprehension*, IEEE, 2011, pp. 219–220.
- [22] J.R. Cordy, The TXL source transformation language, *Sci. Comput. Program.* 61 (3) (2006) 190–210.
- [23] C.K. Roy, J.R. Cordy, NiCad: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, *Proceedings of the Sixteenth International Conference on Program Comprehension*, IEEE, 2008, pp. 172–181.
- [24] C.K. Roy, Detection and analysis of near-miss software clones, (2009) Ph. D. Thesis.
- [25] E.P. Antony, M.H. Alalfi, J.R. Cordy, An approach to clone detection in behavioural models, *Proceedings of the Twentieth Working Conference on Reverse Engineering*, (2013), pp. 472–476.
- [26] S. Melnik, H. Garcia-Molina, E. Rahm, Similarity flooding: A versatile graph matching algorithm and its application to schema matching, *Proceedings of the Eighteenth International Conference on Data Engineering*, IEEE, 2002, pp. 117–128.
- [27] Ö. Babur, L. Cleophas, M. van den Brand, Hierarchical clustering of metamodels for comparative analysis and visualization, *Proceedings of the Twelfth European*

- Conference on Modelling Foundations and Applications, (2016), pp. 3–18.
- [28] C.D. Manning, P. Raghavan, H. Shtze, *Introduction to Information Retrieval*, Cambridge University Press, 2008.
- [29] A.K. Jain, R.C. Dubes, *Algorithms for Clustering Data*, Prentice-Hall, Inc., 1988.
- [30] C.D. Manning, H. Schütze, *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.
- [31] Ö. Babur, L. Cleophas, Using n-grams for the automated clustering of structural models, *Proceedings of the Forty-third International Conference on Current Trends in Theory and Practice of Computer Science*, (2017), pp. 510–524.
- [32] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014. <http://www.R-project.org/>.
- [33] Ö. Babur, L. Cleophas, T. Verhoeff, M. van den Brand, Towards statistical comparison and analysis of models, *Proceedings of the Forth International Conference on Model-Driven Engineering and Software Development*, (2016), pp. 361–367.
- [34] M. Pawlik, N. Augsten, Tree edit distance: robust and memory-efficient, *Inf. Syst.* 56 (2016) 157–173.
- [35] K. Zhang, D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, *SIAM J. Comput.* 18 (6) (1989) 1245–1262.
- [36] H.W. Kuhn, B. Yaw, The hungarian method for the assignment problem, *Naval Res. Log. Q.* (1955) 83–97.
- [37] M.M. Deza, E. Deza, *Encyclopedia of Distances*, Springer, 2009.
- [38] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, AAAI Press, 1996, pp. 226–231.
- [39] M. Stephan, J.R. Cordy, Model clone detector evaluation using mutation analysis, *Proceedings of the Thirtieth IEEE International Conference on Software Maintenance and Evolution*, (2014), pp. 633–638.
- [40] M. Stephan, M.H. Alalfi, J.R. Cordy, Towards a taxonomy for Simulink model mutations, *Proceedings of the Seventh IEEE International Conference on Software Testing, Verification and Validation*, IEEE, 2014, pp. 206–215.
- [41] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D.M. German, D. Damian, The promises and perils of mining GitHub, *Proceedings of the Eleventh Working Conference on Mining Software Repositories*, ACM, 2014, pp. 92–101.
- [42] R. Hebig, T.H. Quang, M.R.V. Chaudron, G. Robles, M.A. Fernandez, The quest for open source projects that use UML: mining GitHub, *Proceedings of the ACM/IEEE Nineteenth International Conference on Model Driven Engineering Languages and Systems*, (2016), pp. 173–183.
- [43] D.S. Kolovos, N.D. Matragkas, I. Korkontzelos, S. Ananiadou, R.F. Paige, Assessing the use of eclipse MDE technologies in open-source software projects. *Proceedings of the OSS4MDE@MoDELS*, (2015), pp. 20–29.
- [44] Ö. Babur, L. Cleophas, M. van den Brand, Towards distributed model analytics with Apache Spark, *Proceedings of the Sixth International Conference on Model-Driven Engineering and Software Development*, 2018, (2018), pp. 767–772.
- [45] M.S. Uddin, C.K. Roy, K.A. Schneider, Simcad: An extensible and faster clone detection tool for large scale software systems, *Proceedings of the Twenty-first International Conference on Program Comprehension*, IEEE, 2013, pp. 236–238.
- [46] Ö. Babur, L. Cleophas, M. van den Brand, Model analytics for feature models: case studies for S.P.L.O.T. repository, *Proceedings of the of MODELS 2018 Workshops*, Co-located with ACM/IEEE Twenty-first International Conference on Model Driven Engineering Languages and Systems, (2018), pp. 787–792.
- [47] D. Wille, Ö. Babur, L. Cleophas, C. Seidl, M. van den Brand, I. Schaefer, Improving custom-tailored variability mining using outlier and cluster detection, *Sci. Comput. Program.* 163 (2018) 62–84.
- [48] M. Stephan, J.R. Cordy, MuMonDE: a framework for evaluating model clone detectors using model mutation analysis, *Softw. Test. Verif. Reliab.* (2018) e1669.
- [49] K. Stol, P. Ralph, B. Fitzgerald, Grounded theory in software engineering research: a critical review and guidelines, *Proceedings of the IEEE/ACM Thirty-eighth International Conference on Software Engineering*, (2016), pp. 120–131.
- [50] M. Stephan, M.H. Alafi, A. Stevenson, J.R. Cordy, Towards qualitative comparison of Simulink model clone detection approaches, *Proceedings of the Sixth International Workshop on Software Clones*, IEEE Press, 2012, pp. 84–85.
- [51] R. Koschke, Survey of research on software clones, *Proceedings of the Duplication, Redundancy, and Similarity in Software*, (2006).
- [52] J. Chen, T.R. Dean, M.H. Alalfi, Clone detection in MATLAB stateflow models, *Softw. Qual. J.* 24 (4) (2016) 917–946.
- [53] D. Strüder, J. Plöger, V. Acrețoaie, Clone detection for graph-based model transformation languages, *Proceedings of the International Conference on Theory and Practice of Model Transformations*, Springer, 2016, pp. 191–206.
- [54] B. Hummel, E. Juergens, L. Heinemann, M. Conradt, Index-based code clone detection: incremental, distributed, scalable, *Proceedings of the IEEE International Conference on Software Maintenance*, IEEE, 2010, pp. 1–9.
- [55] H. Liu, Z. Ma, L. Zhang, W. Shao, Detecting duplications in sequence diagrams based on suffix trees, *Proceedings of the Thirteenth Asia Pacific Software Engineering Conference*, IEEE, 2006, pp. 269–276.
- [56] C.C. Ekanayake, M. Dumas, L. García-Bañuelos, M. La Rosa, A.H. ter Hofstede, Approximate clone detection in repositories of business process models, *Proceedings of the International Conference on Business Process Management*, Springer, 2012, pp. 302–318.