Effective and efficient model clone detection

Störrle, Harald

Published in:
Software, Services, and Systems

Link to article, DOI:
10.1007/978-3-319-15545-6_25

Publication date:
2015

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Störrle, H. (2015). Effective and efficient model clone detection. In R. De Nicola, & R. Hennicker (Eds.), Software, Services, and Systems: Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering (pp. 440-457). Springer. (Lecture Notes in Computer Science, Vol. 8950). DOI: 10.1007/978-3-319-15545-6_25

Effective and Efficient Model Clone Detection

Harald Störrle

Department of Applied Mathematics and Computer Science
Technical University of Denmark (DTU), hsto@dtu.dk

Abstract. Code clones are a major source of software defects. Thus, it is likely that model clones (i.e., duplicate fragments of models) have a significant negative impact on model quality, and thus, on any software created based on those models, irrespective of whether the software is generated fully automatically ("MDD-style") or hand-crafted following the blueprint defined by the model ("MBSD-style"). Unfortunately, however, model clones are much less well studied than code clones. In this paper, we present a clone detection algorithm for UML domain models. Our approach covers a much greater variety of model types than existing approaches while providing high clone detection rates at high speed.

1 Introduction

Code clones (i.e., duplicate fragments of source code), have been identified as "*a major source of faults, which means that cloning can be a substantial problem during development and maintenance*" (cf. [8, p. 494]). As a consequence, a large body of research has been developed on how to prevent, or spot and eliminate code clones (see [12] and [21] for surveys). The problem with code clones is that they are linked only by their similarity, i.e., implicitly rather than explicitly which makes it difficult to detect them. Therefore, changes like upgrades or patches that are often meant to affect all clones in a similar way, are frequently not applied to all of them uniformly. Therefore, code quality deteriorates, and maintenance becomes more costly and/or error prone. Jürgens et al. report that "*nearly every second unintentionally inconsistent change to a clone leads to a fault*" (cf. [8, p. 494]). Experiences with large-scale models suggest that the phenomenon of clones arises in models in a very similar way to how it does in source code. Deißenböck et al. even consider it "*obvious [that] the same [clone-related] problems also occur [...] in model-based development*" (cf. [5, p. 57]). Consequently, the issue of clones has to be addressed for models, too: "*detecting clones in models plays the same important role as in traditional software development*" to use the words of Pham et al. [18, p. 276]. Observe that it is irrelevant whether the models are used as the primary specification of a system, where production quality code is to be generated from models only (as is frequently the case in the automotive industry), or whether models are used in a more liberal way, informing the software creation process rather than dictating it, as is the case for more traditional domain models.

1.1 Approach

In [23], we have studied the origins of model clones, derived a formal definition of model clones, and developed an algorithm for detecting them. Our approach was the first to address all of UML's 14 sub-languages rather than just a single model or diagram type, and so it is not surprising that it left room for improvement in terms of clone detection quality. In this article, we propose a modified algorithm and new similarity heuristics that improve on our previous results, while maintaining its generality. We also validate our approach far better, including 3 new case studies, and a field test.

1.2 Historical Background

As a young PhD-student under the tutelage of Martin, the author participated in the first UML-conference in Mulhouse. There was a great amount of enthusiasm about an important emerging topic, both in academia and industry, and the excitement spread over to Martin's chair, which of course was in a prime position to pick up the trend. Soon, many people at LMU worked on UML, and to this day, it is an important facet of the work done there.

One of the reasons why academia (and many formal-methods-inspired researchers in particular) picked up UML so readily is that it could be viewed as a visual front end to formal methods which had been less than popular with industry. UML, many researchers hoped, might be a way to bring formal methods to practice in a way that is easier to use. In fact, the author, in a fit of juvenile excitement, proclaimed that it had to be so easy as to push the proverbial button. While Martin was wise enough never to subscribe to the push-button benchmark, he was always keen on finding novel ways to improve software quality, one of them being presented here.

1.3 Paper outline

The remainder of this article is structured as follows. In the next Section, we define the notion of model clone, and provide a taxonomy of clone types. In Section 3 we introduce an algorithm for clone detection and the model fragment similarity heuristics used at its core. In Section 5 we evaluate the effectiveness of our approach, and compare it to the related work in Section 6. Finally, we summarize our approach and draw conclusions.

2 Defining model clones

Probably the biggest problem in model clone detection is defining exactly what a model clone is—just as for code clones (cf. [9]). Fig. 1 shows a small part of a domain model of the "Library Management System" case study (LMS) which we use as our standard research object. The figure shows a part of the LMS information model with two Packages.[1] Fig. 1 shows two alternative views of the models: two class diagrams on the left, and the containment tree on the right. Most UML modeling tools today will allow several visual representations of the same model element, such as the Class "Reservation": it appears in both diagrams but exists only once in the containment tree. Thus, it is not a clone, but just one element appearing in two views. On the other hand, the Class "Book" occurs twice in the containment tree (highlighted by the red arrows). Looking closer, we analyze how similar the two model elements are. Assume that in this case, we find that they are identical in all but their internal identifier, so this model element is certainly a duplicate model fragment, i.e., a model clone. This kind of clone arises frequently in practical modeling, typically as a consequence of restructuring models, or from combining independent contributions that are not properly synchronized.

Requiring duplicate model fragments to be *identical* is clearly no adequate definition for the notion of "model clone". Instead, we propose to define a model clone as a set of model fragments each of which is closed under the containment-relationship and that have a high degree of similarity. Observe that this definition includes clones of all sizes, from individual elements via larger sets of model elements like a Property to large Packages containing entire subsystems. In order to refine this definition, we propose a taxonomy of model clone types. For code clone detection, there is a commonly accepted taxonomy of four types (see e.g. [12, 21, 26]). It can be generalized and adapted to accommodate our observations of natural model clones, as follows.

– **Type A: Exact model clones** are identical up to secondary notation and internal identifiers.
– **Type B: Modified model clones** may have small changes to names (e.g., typos) and other values, plus few additions/removals of parts.
– **Type C: Renamed model clones** may have any change to names, attributes, and parts, plus many additions/removals of attributes and parts.
– **Type D: Semantic model clones** are duplicates in content arising, e.g., from convergent modeling.

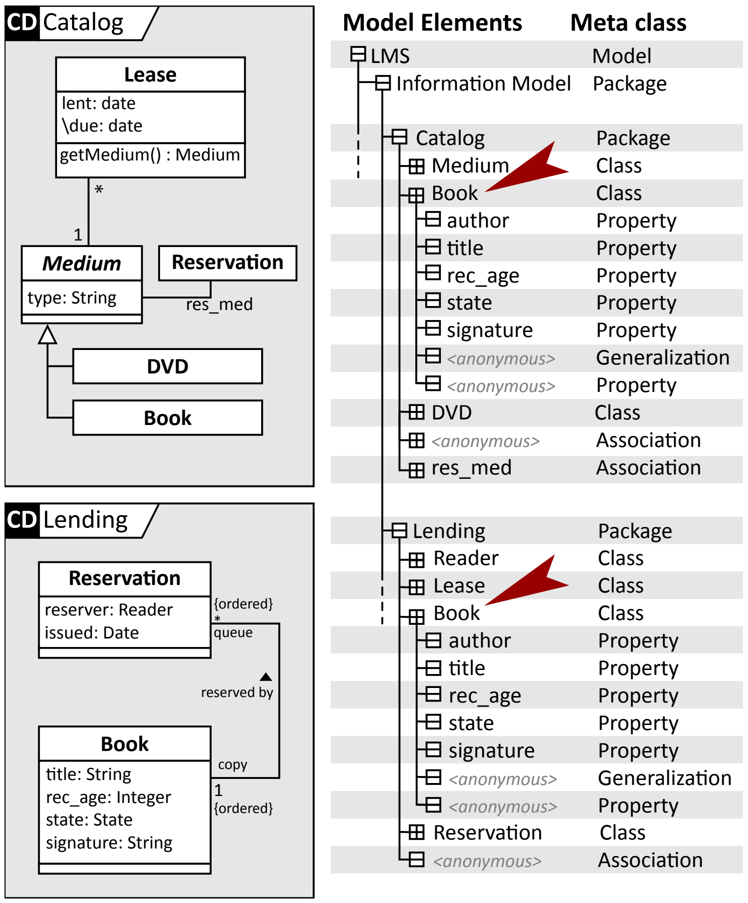
Formal definitions of the notions model, submodel, and clone are found in [23]. Orthogonal to the classification of the kind and degree of changes, model clones can also be classified in other ways.

– **Secondary clones** are pairs of duplicate model elements that satisfy the definition of a clone, but are each parts of larger model fragments, which in turn are clones of each other. For instance, the Class "Book" in the first example is a primary clone, and the Property "author" is a secondary clone.
– **Loophole clones** are duplicates introduced through idiosyncrasies of the modeling language. For instance, any two Activities that refer to the same data item will contain identical copies of a DataFlowNode, because the structure of the UML meta model forces these elements to be contained in an Activity rather than existing on their own. The modeler has no choice but to

[1] We adopt the UML convention of using CamelCaps for UML meta classes.

create such duplicates. In another modeling language (or in a future version of UML), they might not occur.

Both secondary and loophole clones will be among the detected clone candidates, and they may score substantially higher than actual clones, so they are presented before those clone candidates that, arguably, a modeler would expect to be presented first. Clearly, this depends on the similarity measure. For secondary clones, this is easily solved: out of a set of clones that can be (partially) ordered by containment, select only the largest one. But observe that loophole clones can be bigger than true clones. For instance, ActivityPartitions often contain many more elements than Properties, or even Classes. This is a substantial obstacle in similarity scoring, and thus in clone detection.



Model Elements	Meta class
LMS	Model
Information Model	Package
Catalog	Package
Medium	Class
Book	Class
author	Property
title	Property
rec_age	Property
state	Property
signature	Property
<anonymous>	Generalization
<anonymous>	Property
DVD	Class
<anonymous>	Association
res_med	Association
Lending	Package
Reader	Class
Lease	Class
Book	Class
author	Property
title	Property
rec_age	Property
state	Property
signature	Property
<anonymous>	Generalization
<anonymous>	Property
Reservation	Class
<anonymous>	Association

Fig. 1. Duplicate occurrence in diagrams does not constitute a clone (left, class "Reservation"); duplicates in the containment tree *may* be model clones (right, class "Book").

3 Detecting model clones

The starting point of the work reported here is the model clone detection algorithm N2 [23]. We now describe its shortcomings and how we overcome them, resulting in the new algorithm NWS. The improvements are based on (a) the detailed analysis of the algorithms' outputs for a great number of samples, and (b) the systematic exploration of a large number of alternative improvements and settings. We describe the various improvement steps from N2 to NWS.

The basic idea for detecting model clones is straightforward: (1) generate a set of (possibly) matching pairs of model fragments, (2) compute the similarity of each pair, and (3) select those pairs with the highest similarity. The problem with this approach is that, clearly, a model with n elements has up to 2^n fragments, so that naively matching all pairs of fragments would result in exponential runtime. We will now discuss the three stages of the algorithm in turn.

3.1 Model matching

Many previous approaches to model matching have been guided by the intuition that models are more or less graphs, with the added assumption that a large part of the model information is encoded in the links between nodes rather than by the nodes themselves. Following this idea, matching models is essentially finding a subgraph isomorphism which is known to be NP-complete [3].

We believe that this idea is indeed a valid assumption for Matlab/Simulink flow models as considered by much of the related work. We have observed, however, that this intuition does not fit very well with UML models: here, important aspects of the model information are stored in node attributes, e.g. the element names. Furthermore, most of the links between nodes (typically about 85%, see [23]) encode the containment relationship, and thus play a different role. Therefore, looking at models as graphs is somewhat misleading in the case of UML. Instead, we propose to look at models as sets of rich nodes owning small trees, and consider the link structure only in a second step. Considering *only* the containment structure, on the other hand (as is the case with XML-matching), would leave out the semantic information stored in the graph-structure. Also exploiting the symmetry of similarity, we can limit the number of pair-wise comparisons of model elements to $\frac{n^2}{2}$ rather than having to consider all $O(2^n)$ pairs of subgraphs.

Furthermore, in most UML tools it is not possible to change the meta class of a model element once it is created. Thus, creating a clone with a changed type can only be done intentionally or through convergent evolution. We ignore this case and consider only pairs of model elements that are instances of the same meta class. Since a typical UML model contains instances of between 50 and 60 different meta classes, this further reduces the number of fragment pairs to be considered from $\frac{n^2}{2}$ to around $\frac{1}{2}(\frac{n}{50...60})^2$ per element type on average.

Finally, the containment structure of UML models as defined by the UML standard implies that there are many model elements in a model that will typically not be considered as clones by a human modeler, i.e. the loophole clones described in the previous Section. We exclude instances of these meta classes up front, which typically account for more than half of all model elements in a UML model. Thus, the number of pairs to be considered is halved again. Together, these three assumptions drastically reduce the number of clone candidate pairs to be compared in our algorithm. These assumptions are realized in the first part of Algorithm 1.

3.2 Element similarity (comparison and weighing)

In the second step, suitable pairs of elements are compared using different heuristics that are encapsulated in the *sim*-function used in step (2a) of Algorithm 1. The model element similarity function in N2 is based on similarities of the names of elements. We justified this by the observation that element names are very important in domain models. Of course, this notion of similarity is sensitive to renaming, a common operation in domain modeling. Thus, N2 also includes attributes other than the element name. Also, matching of neighbor elements is considered (by types and names).

While experimenting with this approach, we observed that the results were often skewed towards small fragments, because the degree of similarity computed by N2 is normalized by the number of potential similarities. Practically speaking, that means that a pair of fragments that coincide in 3 out of 5 possible ways are assigned a higher similarity measure than a pair of fragments that coincide in 30 of 51 possible ways. However, from a user's perspective, the latter is a more promising clone candidate by far: there is ten times as much evidence for the second pair being a clone than for the first one.

In order to account for this factor, we have implemented a new similarity heuristic in the NWS that includes the "weight" and "binding strength" of clone candidates. The weight which is computed as the number of elements and attributes of the elements contained in the clone candidates normalized by the binding strength. The contained elements are the transitive closure of a model element under UML's containment relationship (meta attribute "ownedMember"). This way, large clones with many small, slightly similar parts may take precedence over smaller clones with high similarity.

3.3 Candidate selection

In practical modeling, clone detection very much resembles a web search: there are many potential hits, but modelers only ever explore a small fraction of them. So, the design goal of model clone detection is to provide the highest possible accuracy in a result set of a given (small) size. While weighing reduces the number of false positive clones, it is vulnerable against the phenomena of secondary and loophole clones we have explained in Section 2. In order to reduce these influences, NWS adds weighing and prioritizing to N2 (see stages 2b and the loop in stage 3 of Algorithm 1, respectively.

Clones of non-trivial size in UML domain models usually imply the existence of very similar sub-fragments, i.e., secondary clones. For instance in Fig. 1, a result set might contain a reference to class "Book" as well as to the property "author" it contains. This can happen despite weighing due to the large variety of similarities and sizes of model clones: the secondary clones of one original may be both more similar and larger than the primary clones of another original. Human modelers usually have the insight to group together primary clones and secondary clones belonging to them, but this puts an extra burden on the modeler. In order to reduce this burden, we explicitly remove secondary clones from the result set (see the comment "case distinction in NWS only" in Algorithm 1). Loophole clones, on the other hand, are excluded by simply adding the respective meta classes to the list of types that are not considered when selecting comparison candidates (see parameter *sensitivity* in Algorithm 1).

4 Implementation

We have implemented our approach and integrated it into the MACH toolset [24]. MACH is available in various variants. First, there is stand-alone version with a textual user interface (called "Subsonic") which is available for download from the MACH homepage `www.compute.dtu.dk/~hsto`. Subsonic is also available in a pre-installed virtual machine that can be run remotely without installation or configuration on the SHARE platform `http://fmt.cs.utwente.nl/redmine/projects/grabats/wiki`, see the respective link at the MACH homepage.

Second, we have also provided a web-service based on MACH (called "Hypersonic", see [1]), where users simply upload a model in a web browser and receive a report on the most likely clone candidates. The implementation technology in all MACH variants (including the Hypersonic web server) is SWI Prolog (see `swi-prolog.org`). The web service is publicly available via the MACH homepage (`http://www2.compute.dtu.dk/~hsto/tools/mach.html`).

5 Evaluation

5.1 Samples

The work reported in this paper derives from the author's experience from two very large scale industrial projects. Due to legal and technical constraints, however, we could not use the models from these case studies directly for this paper. In order to evaluate the quality of our approach, we ran our implementation on four sample models created by students as part of their course work.

The first of these models, called LMS, has been created by a team of four students over 10 weeks; it contains 2,781 model elements (before clone seeding) and 74 diagrams. We used this model for exploration and experimenting with our approach. For the validation, we used three different case studies (called MMM, SBK, and HOS, respectively), created by teams of 5 to 6 students each over a period of 7 weeks. Table 1 presents some size metrics of these models. All of them were created using MagicDraw UML 16.9 (see `www.magicdraw.com`).

Algorithm 1: The NWS clone detection algorithm

```
Input:
  – model $M$,
  – result set size $k > 0$,
  – threshold parameter $sensitivity$

Output:

  – $k$ clone candidates (pairs of elements of $M$)

1 - MATCH
$Elements \leftarrow \{e \in M \mid type(e) \in T\{Action, Actor, Class, \ldots\}\}$;
$Candidates \leftarrow \{\langle e_1, e_2\rangle \mid type(e_1) = type(e_2) \wedge e_1 \neq e_2 \wedge \{e_1, e_2\} \subseteq Elements\}$;

2a - COMPARE
$Comparisons \leftarrow \emptyset$;
forall the $\langle e_1, e_2\rangle \in Candidates$ do
    $E_1 \leftarrow$ transitive closure of $e_1$ wrt. ownedMember;
    $E_2 \leftarrow$ transitive closure of $e_2$ wrt. ownedMember;
    %  sim is a new heuristics for NWS
    $s \leftarrow sim(E_1, E_2)$;
    if $sensitivity > 1/s$ then
        $Comparisons \leftarrow Comparisons \cup \langle E_1, E_2, s\rangle$

2b - WEIGH %new in NWS
$Results \leftarrow \emptyset$;
forall the $\langle E_1, E_2, s\rangle \in Comparisons$ do
    $s' \leftarrow s \cdot \frac{|E_1|+|E_2|}{binding(E_1, E_2, Comparisons)}$;
    $Result \leftarrow Result \cup \langle root(E_1), root(E_2), s'\rangle$;

3 - SELECT
sort $Results$ by decreasing similarity;
$Selection \leftarrow \emptyset$;
%prefer primary over secondary clones
while $|Selection| < k$ do
    $Pick \leftarrow$ first element in $Results$;
    forall the $X \in Selection$ do
        if $X$ is contained in $Pick$ then
            $Selection \leftarrow Selection - X$
    $Selection \leftarrow Selection \cup \{Pick\}$;

return $Selection$;
```

Functions

type	:	type of model element (i.e. meta class)
sim	:	heuristic similarity of model elements, different for N2 and NWS
binding	:	binding strength between two fragments relative to a given set of similarities $binding(E_1, E_2, C) = \sum\{s \mid \langle E_1, E_2, s\rangle \in C\}$
root	:	root element of a fragment
$\lvert E\rvert$	:	number of model elements in a fragment

The LMS model was clone seeded by the author, the other models were clone seeded by their respective authors (i.e., teams of graduate students) as part of a challenge to create clones that our tool could not detect. Identification of seeded clones was achieved through a model difference. A typical example of a seeded Type A clone would be class "Book" in Fig. 1.

Table 1. The sizes of the sample models after seeding

Model	MMM	SBK	HOS	LMS	Sum
Elements	837	1,037	1,650	2,893	6,417
Attributes	2,097	2,915	10,493	17,196	32,701
Diagrams	26	54	33	74	191
Activity	10	27	9	36	82
Use Case	9	21	8	27	65
Class	4	4	7	7	22
Other	3	2	6	8	19
Diagram Types	6	5	8	6	

5.2 Method

As we have discussed in Section 1, there is no undisputed and precise definition of what is and is not a model clone. Relying on industrial models with natural clones, we have no control over the kinds and numbers of clones in them. Since our main objective is to develop algorithms, we resorted to manually seed models with clones. To do so, we randomly picked three typical examples of each of the meta classes UseCase, Class, and Activity in the sample model. We copied them (and their contained model elements), and changed them to emulate Type A, B, and C model clones. Then we marked both the nine original model fragments and the nine copied (and modified) model fragments manually as originals and clones, respectively. This resulted in 145 model elements being marked as clones and 155 being marked as originals, out of a total of 2893 model elements in the model after seeding, i.e., approx. 5.5% of the model elements were marked. A manual inspection of the LMS model did not reveal any natural model clones.

Our annotation allows automatic computation of precision and recall with respect to the seeded clones. The annotation was done by attaching comments to the elements. This way, the elements as such were not changed, as the connection between an element and its comments is established by a link in the comment, not in the commented element. Thus, we can exclude any influence on the clone detection by the annotation. Initially, we ran the clone detection algorithm without restricting the selection, thus yielding a very long list of clone candidates that contained a mixture of seeded clones, natural clones, and false positives. In order to identify the natural clone candidates, we manually reviewed them;

Table 2. Measurements of clone detection quality: each box represents an individual seeded clone, a black box indicates detection within the respective constraints.

Heuristic	N2				NWS			
Results	@10	@20	@30	@100	@10	@20	@30	@100
Precision (seeded)	100.0%	54.5%	37.5%	16.7%	85.7%	53.8%	38.9%	13.3%
Recall (seeded)	44.4%	66.7%	66.7%	66.7%	66.7%	77.8%	77.8%	88.9%
F measure (seeded)	61.5%	60.0%	48.0%	26.7%	75.0%	63.6%	51.9%	23.1%
Type A Clones	■■□	■■■	■■■	■■■	■■■	■■■	■■■	■■■
Type B Clones	■■□	■■■	■■■	■■■	■■■	■■■	■■■	■■■
Type C Clones	□□□	□□□	□□□	□□□	□□□	□■□	□■□	□■■

almost all of them were loophole clones. We then annotated them so that they could be automatically classified by the test instrumentation of our tool. In order to control for bias originating from seeding by the experimenter, we conducted a second experiment. We challenged our students in a modeling class to seed their models with clones that our approach would not detect. This resulted in three clone-seeded models (SBK, HOS, MMM) which were comparable in terms of size and structure to the LMS model (see Table 1 for size metrics of these models). Subsequently, we ran the three detection algorithms on these models.

5.3 Data

Table 2 shows results for the heuristics N2 and NWS with varying result-set sizes. The first two lines show the precision and recall rates as percentages (based on seeded clones only). Since the LMS model did not contain natural clones prior to seeding, we compute recall and precision based on the seeded clones alone. The next three lines show the detection for different kinds of clones. Every box represents a particular seeded clone: the first box is a cloned UseCase, the second one is a cloned Class, and the third one is a cloned Activity. If a box is filled, the respective clone has been detected in the respective result set. So, for instance, □■□ in line "Type C" and column "NWS @30" means that neither the seeded UseCase nor Activity clones of type C were detected by algorithm NWS within the first 30 results, but the seeded Class clone was correctly identified. Similarly, ■■□ in row "Type A" and column "N2 @10" means that the N2 algorithm did not detect an identical copy of an Activity among the first ten results.

Table 3 shows the clone detection results in the models that were seeded by the students. We inspected the models manually to check detection results for accuracy. We also manually classified the clones according to our taxonomy; interestingly, the students' models also contained a number of natural clones that the students apparently were not aware of, but which our tool detected. See Table 3 for the detection rates.

Table 3. Clone detection accuracy by case study and model type: N stands for natural clones, precision and recall are given relative to the first ten results, computed on seeded as well as natural clones.

	Detected/Seeded Clones				NWS @10		
Model	**A**	**B**	**C**	**Natural**	**Precision**	**Recall**	**F-Measure**
MMM	4/ 4	2/ 2	1/ 4	1/-	80.0%	88.9%	84.2%
SBK	6/ 6	2/ 2	0/ 2	2/-	100.0%	83.3%	90.9%
HOS	3/ 3	1/ 2	2/ 5	2/-	80.0%	80.0%	80.0%
All	13/13	5/ 6	3/11	5/-	86.7%	84.1%	85.4%

5.4 Observations

Table 2 shows that precision decreases when recall increases, as is to be expected. Also, Type C clones are less often discovered than Type A and B clones. This is also no surprise since Type C clones have the greatest difference to the originals, and thus the least similarity. The table also shows that NWS provides better detection rates than N2: among the first 10 hits, it covers more seeded Type A and Type B clones than N2. Among the first 20 hits, NWS yields fewer false positives. The same is true when extending the search focus to the first 30 hits. Then, most noticeably, NWS also finds the first Type C clone. Extending the search focus even further to the first 100 hits gives the same result. The increase in the number of false positives indicates that less duplicates are reported. The second Type C clone is reported among the first 100 hits.

Clearly, the results reported so far could have been achieved by tuning the algorithm to fit to the data, in particular to the seeding process. In order to ensure this is not biasing the results in a misleading way, we repeated the clone detection experiments with the models SBK, HOS, and MMM. They present a greater variety of models, and the seeding was done by students, not the author, with the specific instruction to try and break the approach. Even in these samples, however, we found the same differences in the detection rates of different clone types. Similar to the results obtained for the LMS model, Type A and B clones were detected reliably by NWS, that is, all seeded clones with no or little changes were among the first ten clone candidates. In the three models MMM, SBK, and HOS together, three out of eleven Type C clones were also correctly identified. Five natural (i.e., non-seeded) clones were identified, four of which were type A clones, and one of which was a type B clone.

In Fig. 2 (left), we show the first ten hits for each combination of the three algorithms and four models we have studied. We have sorted these ten reported clone suspects by the following four conditions: primary, secondary, duplicate, and false positive. Clearly, the goal is to have as many of the first kind in the result set as they will lead the modeler directly to a clone. Finding a secondary clone is second best, as it does lead the modeler to a clone, but only after having lead the modeler to some suspicious fragment of the clone first. Increasing either of these groups increases the detection precision and recall. Duplicate detections of clones do not add to the set of true positives, thus they do not increase precision and recall, though they still are, technically speaking, correctly identified clones. Finally, false positives are clearly the least desirable kind of reported clone candidates. The perfect score is to have ten primary clones among the first ten reported clone candidates.

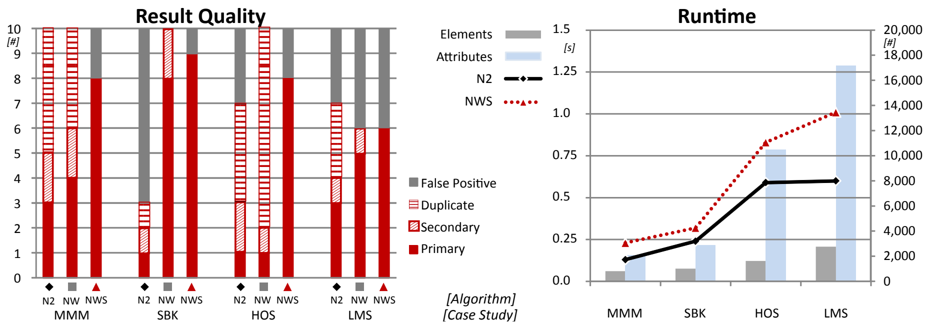


Fig. 2. Performance of three approaches to clone detection: a closer look at the first ten candidates (left); run-time vs. model size (right).

Obviously, the relative difference in clone detection accuracy between the three approaches that we have found in the previous experiment can be observed again in this sample: NWS outperforms N2 in all samples, if not in terms of precision then in terms of a higher rate of primary clones (case studies SBK and LMS). It is also clear, that the different case studies resulted in very different detection rates. Judging by these samples, our approach performs better on clone seeding done by other persons than the author.

Another important aspect of clone detection is the run-time. We have shown the measurements in Fig. 2 (right) by lines. We show the average of three subsequent runs to cancel out any effects due to garbage collection and similar factors. All of the experiments were conducted on a modest laptop computer (Intel i5-2520M 64bit processor at 2.5GHz with 8GB RAM running Windows 7). The run-time differed only insignificantly between NWS and N2 and seems to be independent from the size of the result set (see last row of Table 2). For N2, the run-time seems to slightly increase with the result size, but more detailed measurements would be required to support any stronger claims. The detection run-times are generally very low. To assess the relationship between run-times and model sizes, we have added the number of model elements and attributes in models as grey and blue bars, respectively. The measurements indicate that run-times of all algorithms are mildly polynomial in the model size. In fact, the polynomial appears to be so small, that for the model sizes at hand, it appears to be little more than linear, implying that the approach scales well and is applicable to real models.

5.5 Interpretation of findings

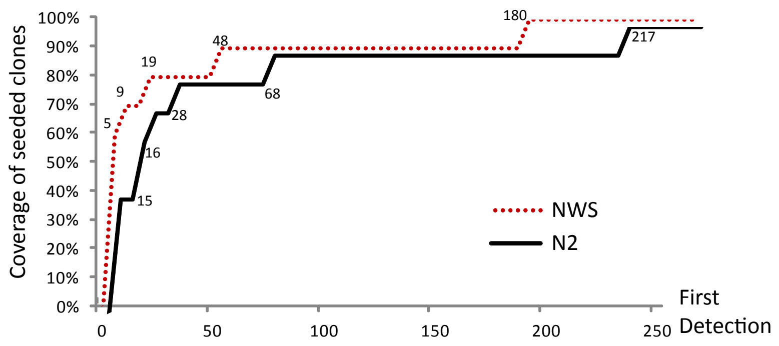


Fig. 3. Both heuristics find all seeded clones eventually, but NWS finds them faster than N2. The numbers in the graph indicate the rank of the last five detected seeded clones for NWS and N2, respectively.

The improvements of NWS over N2 come as no surprise: large duplicates are preferred over small duplicates with the same similarity. However, the details of the detection quality of NWS shown in Table 2 seem to be counter-intuitive: more false positives, and yet a higher coverage rate of seeded clones. This is entirely explained by the specific contribution of NWS, namely, the elimination of secondary clones. When there are indeed secondary clones for one primary clone in a given result set, removing all the secondary clones will promote the next batch of even less likely clone candidates into the result set. Sometimes, this batch contains another, true positive primary clone, but most of the time, there are just more false positives. And so, the precision drops. What NWS does in comparison to N2 is that it compresses the result set towards the top of the list, i.e., the quality of the first hits is improved. This can be seen quite clearly in Fig. 3, where we ran the three heuristics again and kept increasing the result set until all of them had perfect recall. We recorded the earliest position where each of the seeded clones was detected (x-axis) and plotted these against the number of detected clones in terms of the coverage (y-axis). It is easy to see how NWS consistently finds the seeded clones earlier than the other two heuristics.

Considering the run-time, it is at first sight surprising that the more elaborate heuristics in NWS would run faster than that of N2. However, recall that the largest part of the run-time is determined by model size, and that more selective similarity heuristics also imply an earlier elimination of potential solutions, reducing resource consumption. It is difficult to compare other approaches in terms of run-time: different settings may strongly influence the results. It does seem to be true, though, that competing approaches generally have higher run-times, that are either in the same order of magnitude (eScan), or one to three orders of magnitude larger (CloneDetective and aScan, respectively, see [18, p. 285]).

5.6 Threats to validity

We have argued that code clones are actually occurring in practical settings, and that they are potentially damaging. However, most of our argument is only based on plausibility and subjective observations. Also, since this is a new area of research, there is not yet a large body of literature on this topic we can refer to support our point of view. Roy & Cordy described this as: "*more empirical studies with large scale industrial and open source software systems are required.*" (cf. [21, p. 87]). However, it is very difficult to get access to industrial models, and there are very few suitable freely available models, a problem that impedes progress in this field (cf. the "Free Models Initiative", [25]).

The generalizability of our findings is limited by the number and the nature of the models used to develop and validate our approach and the nature of the clones in them: First, the model sample was not created in an industrial context, but in an academic environment, so the models may not be representative. Second, the clones in the sample models are not natural but seeded, i.e. artificial, so they may not be representative of the phenomena found in real models.

With regards to the first argument, consider that the related work in this area has similar limitations: while they may use models of industrial origin, they use very small sets of such models: e.g., the validations of [8], [6], and [18] are based on five, one, and four different models, respectively. Clearly, such small samples do not exhibit a higher level of representativeness than our models do. In the absence of large scale representative field studies, using "real" models cannot claim higher validity than using seeded models—only a large scale field study will allow more general conclusions. However, in vitro work such as presented in this paper is a necessary step towards such a large scale field study.

This observation applies to the second argument, too: seeded clones might not be representative of real clones, but using such specimen is a necessary stepping stone while better sample models are missing. Moreover, by seeding the clones manually we can ensure that all kinds of clones are present in defined quantities and qualities. In natural models, such properties are rarely found, and any such selection would of course introduce undue bias, thus threatening the representativeness of the model sample again. Since the primary purpose of the work reported in this article is to develop algorithms, however, we think manual seeding with full control over quantity and quality of clones across all categories is not just acceptable, but actually essential. Developing our approach with the models seeded by students would have been much more difficult, as the detection results in Fig. 2 (left) suggest.

Still, one might object that it is unacceptable if the seeding is done by the author himself; clearly, he is a potential source of bias. Therefore we also conducted the second experiment where we had no control over the models or the clone seeding process. Surprisingly, the detection rates there are *better* than for the models under the control of the authors, suggesting that the original benchmarks were biased, but no in favor of the algorithm under test, but against it. As we have remarked, the resulting clone seedings were indeed different from what we had expected, sometimes in quite surprising ways. However, our system recognized 18 out of 18 seeded Type A and B clones, and 3 out of 11 seeded Type C clones, within the very low threshold of just ten candidates. The seeded clones that were not among the top ten candidates had undergone substantial changes that made them hardly recognizable as clones, even to human observers (in some cases, this included the students that created the respective clones).

5.7 End User Evaluation

The original implementation of our approach [23] had well over 100 parameters to be set manually. It required a deep understanding of the algorithms limiting the audience. Therefore, we have integrated our approach into the MACH model analysis and checking tool. We then deployed MACH to an undergraduate course on model based software development (41 students). After the course, we surveyed the students for their tool usage during the course (68% response rate), and found that students had some trouble installing MACH, and were unused to command line interfaces as such, but there were no negative remarks on the clone detection facilities. However, there were several positive remarks about this feature, e.g., students reported that it had helped them assess the quality of their models in unexpected ways. There were no problems in interpreting the results of clone detection either, although these results were not always perfect.

The field test clearly demonstrated, that it is possible to empower students with a very low level of qualification to routinely run an advanced clone detection algorithm without any additional support, without any noticeable problems. It is quite telling that the most negative comment on the clone detection was that "*at some point it reported clones that were actually not clones*" (i.e., false positives). We have since then used MACH in two more classes (48 and 54 students, respectively), without any problems.

6 Related work

There is a large body of work on code clones: [12] provides a survey of the field, and [4] gives an overview of the state of the art. Clones in models, on the other hand, have received much less attention, only in the last few years have there been investigations into this topic. They can be divided broadly into four classes.

First, the CloneDetective system by Deißenböck et al. detects clones in Matlab/Simulink flow graph models [6, 5]. This approach suffers from "*a large number of false positives*" (cf. [6, p. 609]), as the authors admit. It is also relatively slow (see [18]), since it effectively uses a graph isomorphism algorithm, Pham et al. [18] report run-times for CloneDetective in the range of a few hundreds of seconds for non-trivial models. Pham et al. then address this shortcoming with their ModelCD system using a hash-based clone detection algorithm. They achieve run-times roughly comparable to the ones we have presented above. Both CloneDetective and ModelCD are limited to Matlab/Simulink flow-models.

Second, there are various approaches dealing with matching of individual UML model types such as interactions [13, 20] or state charts [17]. In contrast, our approach deals with all the UML's notations, including flow-like models such as activities, but also class models, use case models, interactions ("sequence diagrams"), and state machines.

Third, there have been approaches that have explored graphs and graph grammars as a generic underlying data structure for all types of models (cf. the PROGRESS system, [16, 22]). These approaches have developed graph matching algorithms that might possibly be used for clone detection, but have not been studied under this angle. It does not seem like a promising avenue to explore, however, due to the fact that UML models do not store (much of) their semantic information in a graph structure. Rather than relatively dense and homogeneous networks of light-weight nodes, UML models are trees of heavy-weight nodes with some additional non-tree connections. Generic graph algorithms do not exploit this fact and thus miss a valuable opportunity (see [19] for a survey of graph- and tree-matching algorithms). In particular, consider Similarity Flooding (SF) [14], which is a fixed point computation that may take many iterations. Given the large number and size of potential mappings between duplicate fragments, such algorithms will not be applicable to clone detection for realistic models. To use the words of the inventors of Similarity Flooding: "*This approach [Similarity Flooding] is feasible for small schemas, but it does not scale to models with tens of thousands of concepts.*" (cf. [15, p. 3]). The heuristics we propose, however, appear to scale almost linearly. Moreover, Similarity Flooding depends on a reasonable initial seed value which is available for model matching in version control, but not for the kind of matching task we find in model clone detection.

Fourth, there are approaches that explore model matching for version control of models. Alanen and Porres [2] study set theory-inspired operators on models. Kolovos et al. on the other hand have proposed the Epsilon Merge Language ([11]) using a identifier-based matching process, while Kelter et al. [10] uses the Similarity Flooding algorithm in their SiDiff tool. Observe that in version control one can reasonably expect most model elements to have the same unchanged internal identifier in two subsequent model versions. Thus, it is easy to find a high-quality mapping to seed a matching algorithm. In clone detection, however, the problem is to efficiently find the mapping in the first place.

7 Conclusion

Model clones increasingly are a problem for model based development: there is "*strong evidence that inconsistent [code] clones constitute a major source of faults*" (cf. [8, p. 494]) and "*detecting clones in models plays the same important role as in traditional software development*" (cf. [18, p. 276]). However, there is currently not much published work on model clones, in particular on clones in UML models. In [23], we have developed a clone type taxonomy, and proposed an algorithm to detect clones. In this paper, we improved our earlier algorithm in terms of detection quality, and provide new front-ends to our implementation so that it can be used by non-experts. We also improved the scientific validity of our results by testing our approach with additional case studies that were clone seeded by independent parties, and a field test to assess the practical usability of our tool and approach.

The published data on approaches such as ModelCD and CloneDetective is somewhat incomplete making it difficult to compare them, though it seems that our approach is at least as good in terms of run-time and detection quality, while being applicable to a far wider range of model types: existing approaches cover only a single model type (e.g., UML State machines, or Matlab/Simulink models), while our approach applies to *all* of UML, and even DSLs.

Improving our previous work, the NWS algorithm provides much better detection rates, in particular with respect to improving the ranking of the first few clone candidates. Thus, from a modeler's point of view, the findings presented by NWS are of much higher quality. We have evaluated our approach, including also a field test with undergraduate students, underlining that clone detection is a practical tool rather than a mere research prototype. Reducing the number of false positives was made possible by understanding the structure of clones; these insights will likely be applicable in use cases of model similarity, too.

References

1. Acretoaie, V., and Störrle, H. Hypersonic - Model Analysis as a Service. In *Joint Proc. MODELS 2014 Poster Session and ACM Student Research Competition* (2014), S. Sauer, M. Wimmer, M. Genero, and S. Qadeer, Eds., vol. 1258, CEUR, pp. 1–5. available online at `http://ceur-ws.org/Vol-1258`.
2. Alanen, M., and Porres, I. Difference and Union of Models. In *Proc.* 6th *Intl. Conf. Unified Modeling Language (≪UML≫'03)* (2003), P. Stevens, J. Whittle, and G. Booch, Eds., vol. 2863 of *LNCS*, Springer Verlag, pp. 2–17.
3. Cook, S. A. The complexity of theorem-proving procedures. In *Proc.* 3rd *Ann. ACM Symp. Theory of Computing* (1971), ACM, pp. 151–158.
4. Cordy, J. R., Inoue, K., Koschke, R., and Jarzabek, S., Eds. *Proc. 4th Intl. Ws. Software Clones (IWSC'10)* (2010), ACM.
5. Deissenböck, F., Hummel, B., Juergens, E., Pfaehler, M., and Schätz, B. Model Clone Detection in Practice. In Cordy et al. [4], pp. 57–64.
6. Deissenböck, F., Hummel, B., Schaetz, B., Wagner, S., Girard, J., and Teuchert, S. Clone Detection in Automotive Model-Based Development. In *Proc. IEEE 30th Intl. Conf. Software Engineering (ICSE)* (2008), IEEE Computer Society, pp. 603–612.
7. Proc. IEEE 31st Intl. Conf. Software Engineering (ICSE). In *Proc. IEEE 31st Intl. Conf. Software Engineering (ICSE)* (2009), IEEE Computer Society.
8. Juergens, E., Deissenböck, F., Hummel, B., and Wagner, S. Do code clones matter? In ICSE'09 [7], pp. 485–495.
9. Kapser, C., Anderson, P., Godfrey, M., Koschke, R., Rieger, M., Van Rysselberghe, F., and Weissgerber, P. Subjectivity in clone judgment: Can we ever agree? Tech. Rep. 06301, Internationales Begegnungs- und Forschungszentrum für Informatik Schloß Dagstuhl, 2007.
10. Kelter, U., Wehren, J., and Niere, J. A Generic Difference Algorithm for UML Models. In *Proc. Natl. Germ. Conf. Software-Engineering (SE'05)* (2005), K. Pohl, Ed., no. P-64 in Lecture Notes in Informatics, GI e.V, pp. 105–116.

11. Kolovos, D. S., Paige, R. F., and Polack, F. A. Merging models with the Epsilon Merging Language (EML). In *9th Intl. Conf. Model Driven Engineering Languages and Systems (MoDELS'09)* (2006), O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., no. 4199 in LNCS, Springer Verlag, pp. 215–229.
12. Koschke, R. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software* (2006), A. Walenstein, R. Koschke, and E. Merlo, Eds., no. 06301 in Dagstuhl Seminar Proceedings, Intl. Conf. and Research Center for Computer Science, Dagstuhl Castle.
13. Liu, H., Ma, Z., Zhang, L., and Shao, W. Detecting duplications in sequence diagrams based on suffix trees. In *13th Asia Pacific Software Engineering Conf. (APSEC)* (2006), IEEE CS, pp. 269–276.
14. Melnik, S., Garcia-Molina, H., and Rahm, E. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proc. 18th Intl. Conf. Data Engineering (ICDE'02)* (2002), IEEE, pp. 117–128.
15. Mork, P., and Bernstein, P. A. Adapting a Generic Match Algorithm to Align Ontologies of Human Anatomy. In *Proc. 20th Intl. Conf. Data Engineering (ICDE'04)* (2004), IEEE Computer Society, pp. 787–791.
16. Nagl, M., and Schürr, A. A Specification Environment for Graph Grammars. In *Proc. 4th Intl. Ws. Graph-Grammars and Their Application to Computer Science* (1991), H. Ehrig and G. Kreowski, H.and Rozenberg, Eds., vol. 532 of *LNCS*, Springer Verlag, pp. 599–609.
17. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. Matching and merging of statecharts specifications. In *Proc. 29th Intl. Conf. Software Engineering (ICSE)* (2007), IEEE Computer Society, IEEE Computer Society, pp. 54–64.
18. Pham, N. H., Nguyen, H. A., Nguyen, T. T., Al-Kofahi, J. M., and Nguyen, T. N. Complete and accurate clone detection in graph-based models. In ICSE'09 [7], pp. 276–286.
19. Rahm, E., and Bernstein, P. A. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal 10* (2001), 334–350.
20. Ren, S., Rui, K., and Butler, G. Refactoring the scenario specification: A message sequence chart approach. In *9th Intl. Conf. Object-Oriented Information Systems* (2003), no. 2817 in LNCS, Springer, pp. 294–298.
21. Roy, C. K., and Cordy, J. R. A Survey on Software Clone Detection. Tech. Rep. TR 541, Queen's University, School of Computing, 2007.
22. Schürr, A. Introduction to PROGRESS and an Attribute Graph Grammar Based Specification Language. In *Proc. 15th Intl. Ws. Graph-Theoretic Concepts in Computer Science (WG'89)* (1989), M. Nagl, Ed., vol. 411 of *LNCS*, Springer Verlag, pp. 151–165.
23. Störrle, H. Towards Clone Detection in UML Domain Models. *J. Softw. Syst. Model. 12*, 2 (2013), 307–329.
24. Störrle, H. UML Model Analysis and Checking with MACH. In *4th Intl. Ws. Academic Software Development Tools and Techniques* (2013), M. van den Brand, K. Mens, P.-E. Moreau, and J. Vinju, Eds.
25. Störrle, H., Hebig, R., and Knapp, A. The Free Models Initative. In *Joint Proc. MODELS 2014 Poster Session and ACM Student Research Competition* (2014), S. Sauer, M. Wimmer, M. Genero, and S. Qadeer, Eds., vol. 1258, CEUR, pp. 36–40.
26. Tiarks, R., Koschke, R., and Falke, R. An Assessment of Type-3 Clones as Detected by State-of-the-Art Tools. In *Intl. Ws. Source Code Analysis and Manipulation* (2009), IEEE Computer Society, pp. 67–76.