Towards Clone Detection in UML Domain Models

Störrle, Harald

Published in:
Software and Systems Modeling

Link to article, DOI:
10.1007/s10270-011-0217-9

Publication date:
2013

Citation (APA):
Störrle, H. (2013). Towards Clone Detection in UML Domain Models. *Software and Systems Modeling*, *12*(2), 307-329. https://doi.org/10.1007/s10270-011-0217-9

Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Towards Clone Detection in UML Domain Models

Harald Störrle

Department of Informatics and Mathematical Modeling (IMM)
Technical University of Denmark (DTU)

Received: 2010-11-21 / Revised version: 2011-07-03 / Final version: 2011-09-12

Abstract Code clones (i. e., duplicate fragments of code) have been studied for long, and there is strong evidence that they are a major source of software faults. Anecdotal evidence suggests that this phenomenon occurs similarly in models, suggesting that model clones are as detrimental to model quality as they are to code quality. However, programming language code and visual models have significant differences that make it difficult to directly transfer notions and algorithms developed in the code clone arena to model clones.

In this article, we develop and propose a definition of the notion of "model clone" based on the thorough analysis of practical scenarios. We propose a formal definition of model clones, specify a clone detection algorithm for UML domain models, and implement it prototypically. We investigate different similarity heuristics to be used in the algorithm, and report the performance of our approach. While we believe that our approach advances the state of the art significantly, it is restricted to UML models, its results leave room for improvements, and there is no validation by field studies.

1 Introduction

Code clones (i. e., duplicate fragments of source code), have been identified as a major source of software quality issues over the last years [9]. As a consequence, a large body of research has been developed on how to prevent, or spot and eliminate code clones (see [14] and [26] for excellent surveys). The problem with code clones is of course 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 applied only to one or a few of them. Other clones may accidentally remain unchanged, and so, code quality deteriorates, and maintenance becomes more costly and/or error prone.

Domain models—also known as conceptual or analysis level models—are commonly used in the IT industry, for example in the early phases of large software development projects, schema integration for databases, and business process management and optimization. Proponents of the Model-Driven Development (MDD) paradigm go further and claim that modeling will eventually replace programming; see for instance Selic's assertion in [29, p. 19]: "*MDDs defining characteristic is that software developments primary focus and products are models rather than computer programs*". Even more clearly, the Model-Driven Architecture (MDA, [22]) attempts to "*shift the focus of software development away from the technology domain toward the ideas and concepts of the problem domain*", as proclaimed by Booch, Rumbaugh and others in the "MDA Manifesto" (see [2, p. 3]). Whether one does believe in this vision or not, there can be no doubt that models play an important role in many industrial development activities of many organization today, and that models can reach quite substantial sizes (cf. [31,34]).

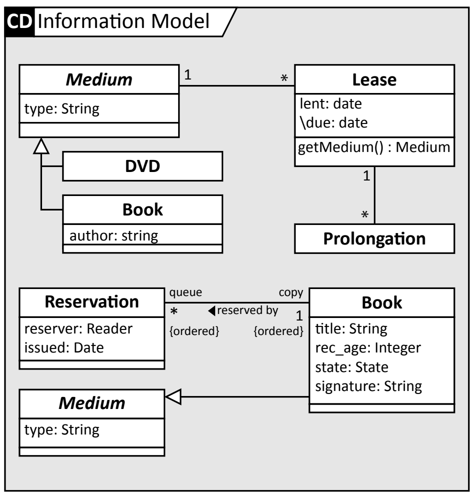
If programs are indeed progressively being replaced by models, it comes as no surprise that phenomena known from source code occur in models, too. Indeed, experiences with large scale domain models suggest, that the phenomenon of clones arises in models in a very similar way than it does in source code. Deissenbck et al. even consider it "*obvious*" that "*the same [clone-related] problems also occur [...] in model-based development*" (cf. [5, p. 57]). Consequently, the clone issue 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 [23].

Paraphrasing Baxter[1], we tentatively define a *model clone* as a set of similar or identical fragments in a model. Obviously, this is a rather unsatisfying definition in many ways, but for the time being, we adopt it as our working definition. Informed by the related work on code clones, we focus on four major challenges:

1. understand the structure of real clones (Section 2) and derive a practical definition of model clones from our findings (Section 3);
2. quantitatively analyze the structure of medium to large scale models and develop heuristics informed by the analysis results (Section 4);
3. derive a formal framework for model clones and develop an algorithm detect clones in models of realistic size and structure (Section 5);
4. implement the algorithm and heuristics (Section 5.3), balancing precision and recall against acceptable run time (Section 6).

We conclude with a survey of the related work (Section 7) and a discussion of our results (Section 8.1). Previous work leading to this article has been published as [35].

[1] "*Clones are segments of code that are similar according to some definition of similarity*" op cit. [14, p. 2].



Model Elements	ID	Meta class
⊟ LMS	0	Model
⊟ Catalog	3	Package
⊞ Medium	4	Class
⊞ Book	8	Class
⊞ DVD	12	Class
⊞ Rating	16	Enumeration
⊞ State	22	Enumeration
⊞ BoardGame	27	Class
⊞ -	28	Association
⊞ Search Catalog	30	Package
⊞ Update Catalog	49	Package
⊟ Lending	69	Package
⊞ Medium	70	Class
⊞ Lease	75	Class
⊞ Prolongation	81	Class
⊞ Reservation	88	Class
⊞ -	92	Association
⊞ Lease Medium	95	Package
⊞ Prolong Lease	152	Package
⊟ Reserve Medium	214	Package

Fig. 1 UML Models have two parallel structures: an external, visual representation as diagrams (left); and an internal, tree-like structure (right).

2 Analyzing model clones

We will use the Library Management System (LMS) case study as our running example, a small analysis-level model expressed in UML 2.2. The LMS consists of two subsystems represented as packages named "Catalog" and "Lending".Each of them contains Classes, Associations, and sub-Packages encapsulating UseCase and Activity descriptions of business processes.[2] Typical UML tools present such a model using two parallel structures: a set of views or diagrams, and a containment tree. Fig. 1 shows a schematic representation of a part of the LMS highlighting these two structures.

Fig. 1 (right) shows the model proper by way of a containment tree arising from the ownedElement UML meta attribute, Fig. 1 (left) shows both one possible view of the model as a UML class diagram entitled *Information Model* showing a view of elements from both LMS packages.

Looking at the diagram first, we can easily see that there are two occurrences of both the two classes named "Book" and "Medium" in the diagram, which may or may not be clones—just looking at the view cannot answer this question. The two occurrences of the class named "Book" show quite different signatures while the two occurrences of the class named "Medium" coincide in many aspects. Yet, an exhaustive search of the containment tree to the right shows, that there is only one occurrence of a class named "Book". So, the different diagram elements "Book" are just harmless separate views of the same model element "Book", highlighting different properties. On the other hand, there are two occurrences of classes named

[2] We generally write UML meta classes in the way they are written in the UML Standard [21], i. e., using CamelCaps, e. g., Class or UseCase.

"Medium" in the containment tree (identifiers 4 and 70), so "Medium" is definitely a clone candidate. It is still not obvious, though, which reference in the diagram refers to which model element, or whether they refer to the same model element or not. Also, there are two classes in the containment tree (named "BoardGame", identifier 27), which is not referenced in this view. To naïve modelers, who are often equating models and diagrams, this model element will be invisible.

As we have said, most contemporary UML tools provide not just (multiple) diagram views of the model, but also a more basic view of the containment-tree of the model elements themselves. If a tool also provides functions to navigate from one to the other, to identify model elements without diagrammatic representation, and so on, a keen modeler can actually find the clones in the example above manually. However, this is a very tedious process even for small models, and it is limited to clones with identical names. Thus, this is no practical way to detect clones.

2.1 Model clones through copy/paste

It is well known that most code clones are created by ad-hoc-reuse through copy/paste: fragments are copied but no explicit link is established between them. Since this practice is common in modeling, too, it is plausible to assume that the same holds for models. In fact, something worse happens: the two parallel structures discussed above are affected in different ways by editing operations like copy and paste. According to our observations, it is common practice among industrial modelers to interact with the model primarily or only through the diagrams, i. e. the model views rather than through the model proper. This leads to a hazardous feature interaction between common editing commands: copying a diagram element may either create only a new (visual) reference to a pre-existing model element, or also a new copy of the model element. Deleting a diagram element, on the other hand, is usually implemented as deleting only the visual reference, but leaving the corresponding model element untouched. So, a modeler may inadvertently create clones when s/he just wants another visual reference, or when s/he deletes the visual reference only, instead of both the diagram element and the model element. Fig. 2 sketches a typical situation.

There, different configurations of views and models are considered, each of which is shown both through a view (top) and a part of the composition tree (bottom). A typical tool will create both model elements and diagram elements when copying, but will only delete a diagram element when deleting. Thus, starting in configuration 1, a user might copy/paste "Book" to reach configuration 2, thus introducing a (visible) clone (class "Book" with identifier 3). If the modeler then wants to revert the previous operation but erroneously uses "delete" instead of "undo", the diagram element is deleted while the model element is kept. So, the clone has become invisible to a casual modeler, and in models of realistic size, it can only be discovered

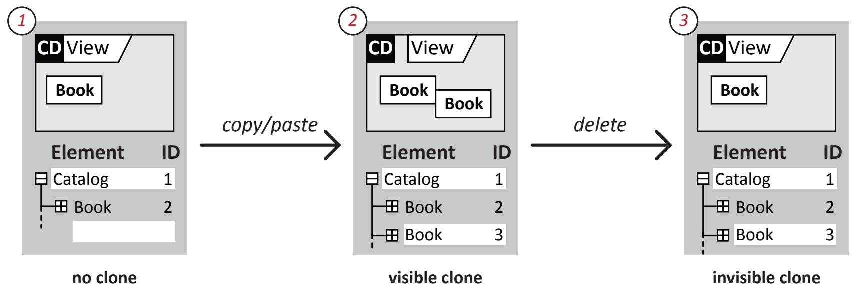


Fig. 2 Simple sequences of editing operations can create discrepancies between the external (top) and the internal structures (bottom) of models.

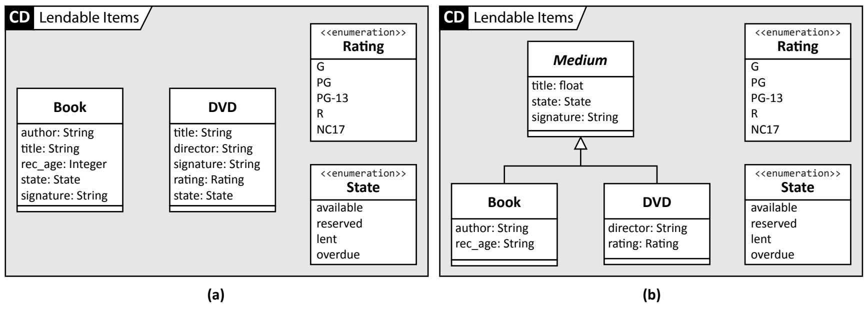


Fig. 3 Clones may indicate unfinished modeling work (1/3): an unfinished class model (left); factoring out commonalities removes clones (right).

again through luck or exhaustive search. Of course, many tools offer different versions of copy and delete, but this still leaves room for individual mistakes. This way, copy/paste of model fragments is even more prone to clone introduction than copy/paste of code fragments.[3]

2.2 Model clones through unfinished modeling

Some clones may be characterized as the remains of unfinished modeling. For instance, in the process of modeling the LMS, we may model different kinds of media of the library, and we may model several attributes repeatedly, see Fig. 3 (a). There are at least three duplicates: properties “title”, “id”, and “state” occur both identically in classes “Book” and “DVD”. They could be factored out into a class “Medium” which is a common superclass to both “Book” and “DVD”, yielding the model shown in Fig. 3 (b).

Other model types share this phenomenon as Figures 4 and 5 demonstrate. In Fig. 4 (a), there are two instances of the transitions with triggers “lease” and “return”, respectively. By introducing a common super state to

[3] Also, some tools avoid this problem altogether by enforcing a one-to-one correspondence between model elements and diagram elements as the default (e. g., ADONIS, cf. [10]), but there are more ways model clones are created.

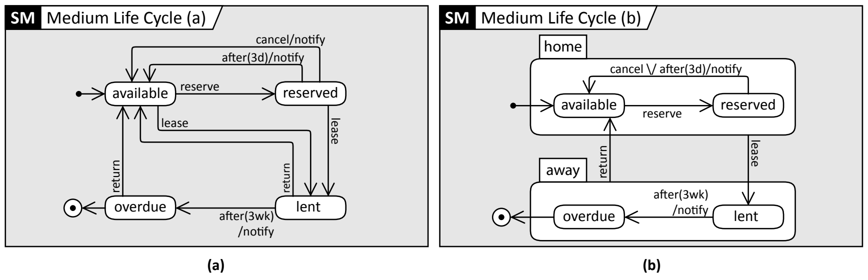


Fig. 4 Clones may indicate unfinished modeling work (2/3): an unfinished state machine model (left); factoring out commonalities removes clones (right).

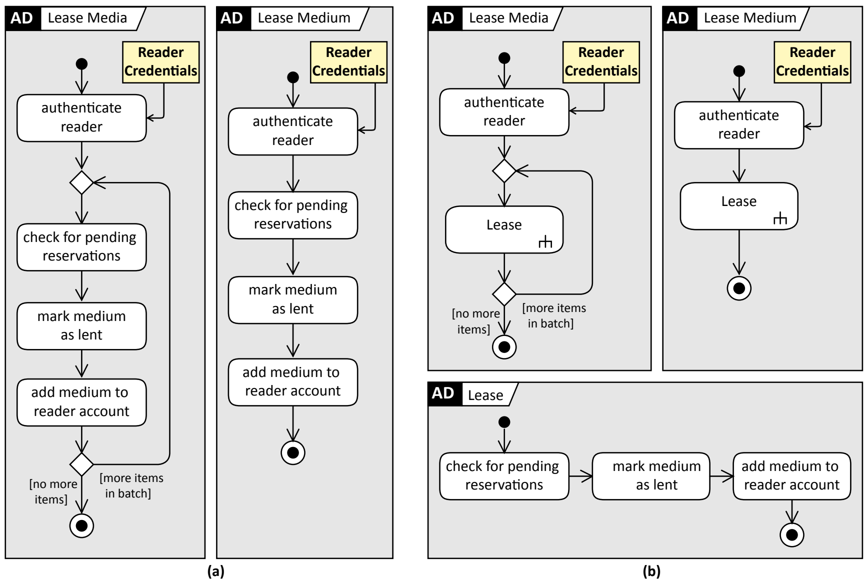


Fig. 5 Clones may indicate unfinished modeling work (3/3): an unfinished activity model (left); factoring out commonalities removes clones (right).

states “lent” and “overdue”, these duplicates can be avoided to yield the state machine shown in Fig. 4 (b). Similarly, there are two transitions with triggers “cancel” and “after(5d)” that share source- and target-states as well as their action. They could be merged as shown in Fig. 4 (b). Similar refactorings can be found in activity diagrams (see Fig. 5), where recurring fragments of activities can be factored out into independent activities (in this example “Lease” in Fig. 5 (b), bottom). Observe, that our discussion explicitly refers to domain models, where we usually consider such duplicates as unwanted redundancies. In design and implementation level models, on the other hand, there may be more reasons to keep such duplicates even in a finalized model.

2.3 Model clones through language loopholes

While the kinds of model clones discussed in the previous two sections have some kind of counterparts in code clones, and are likely to appear in all kinds of modeling languages, the following one is specific for UML-models. It derives from the way the meta model of UML is defined, and the way UML models are stored in XMI. Reconsider Fig. 5, and notice the DataFlow-Nodes named "Reader Credentials". They occur twice, as parts of the two Activities "Lease Media" and "Lease Medium", and there is no way of factoring them out: the UML meta model defines data nodes as being integral part of an activity. The same is true for individual Actions like "authenticate reader", or ActivityPartitions. All of these model elements cannot exist on their own, outside an activity, and thus, they cannot be shared among different Activities. Therefore, they necessarily occur repeatedly in a model.

Another kind of inevitable duplicates specific to UML are the manifold meta model elements that carry little information but occur frequently. Examples are FinalNodes of Activities, Generalizations of Classifiers, and ValueSpecifications of MultiplicityElements. They all can only exist nested is their respective container, and they all occur over and over again in models in largely or completely identical form. A clone detection algorithm must make sure not to present these elements as clones to the user.

Unfortunately, it is not enough to simply disregard some types of model elements, because whether or not a model element is considered as a "proper" modeling concept may depend more on the modeler than on the modeling language. For instance, the Associations in Fig. 1 discussed above could be regarded as secondary elements because they lack attributes. But consider also the Association "reserved by" in Fig. 1 which has many properties, e. g., role names and constraints, a name, and a reading direction which indicate that it this is a model element of some domain significance. Similarly, it is also not enough to disregard "small" model elements, because relevant model elements can be quite small (e. g., Classes with few Features), while "big" elements can be rather lightweight (e. g., MultiplicityElements). Also, elements such as FinalNodes may well be considered meaningful (see Fig. 7).

While the clone types discussed in this section are specific to UML, other modeling languages invariably have their own idiosyncrasies often leading to very similar effects.

2.4 Model clones on purpose

While many duplicate code fragments are doubtlessly defects introduced through mistakes or language loopholes, others might even be introduced on purpose. For instance, assume that two variants of a model are to be created to contrast alternative options, successive development stages, or similarities of competing systems. In those cases, two or more submodes

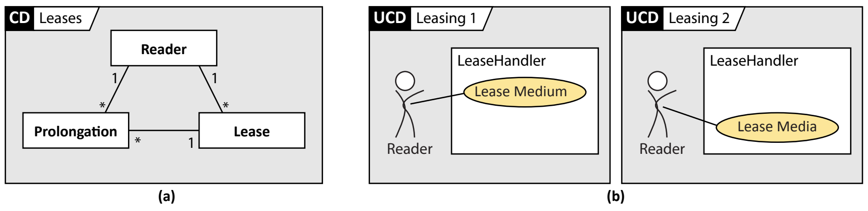


Fig. 6 Small differences of small elements may lead to false positive clone detections: Associations (left), or Use Cases (right) have few attributes to distinguish them.

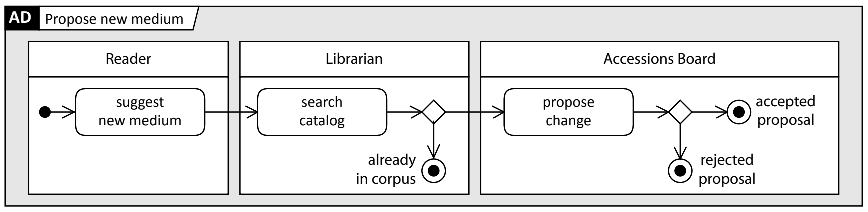


Fig. 7 Even "small" nodes such as FinalNode in Activities can carry significant meaning.

might be created as part of one model. The semantic overlap between the variants can give rise to duplicate model fragments. A similar situation arises if a model contains test cases, which come in great numbers that will frequently show only small differences.

Also, there might be methodological reasons for introducing repetitions, e., g., the common practice to describe business processes through a Use-Case and an Activity using the same name. There, the implicit connection through the shared name is fully on purpose.

3 Defining the notion of "model clone"

Probably the biggest problem with source code clones is defining exactly what is and what is not a clone (cf. [11]). For model clones, this is at least as difficult. Our working definition "*a model clone a set of similar or identical fragments in a model*" is obviously too vague to be very helpful: what is a model and how are they best represented? What is a model fragment, and what properties and size should it have? How should similarity be defined? So, before we can even start to look at ways to spot model clones, we must elaborate our intuition, and improve our working definition of model clone. We start off by looking at the state of the discussion in the code clones community.

3.1 Code clones vs. model clones

If code clones and model clones are indeed comparable as we have argued, it should be possible to transfer (some of) the notions, techniques, and lessons learned from code clone research to the model clones arena. However, past experiences in studying version control of models in contrast to version control of code also suggest that there are significant differences between models and code which may be relevant for clones as well. The differences can be traced down to the following factors.

3.1.1 Language/Tool Integration Programming languages are largely independent of a particular development environment: a Java program created with one IDE can easily be transferred to another, or to a plain text editor. In the end, a program is just a (set of) text file(s). Models, on the other hand, are usually very tightly integrated with some tool, and even if we restrict ourselves to UML, exchanging model data between different modeling tools may be difficult or plain impossible. So, ultimately, when speaking about model clones, it is not sufficient to speak about the modeling language, we may have to take into account the CASE tool used to create the model, too.

3.1.2 Artifact structure The code of a system can easily be represented as a directory tree of text files, each of which can be considered as a long string of characters or tokens. Models, on the other hand, often have a graph-structure stored in a repository, but some tools use other ways of representing models, and since languages and tools are tightly integrated, we have to take tool-specific representations into account to some degree.

3.1.3 Identification In source code, elements like types, procedures, and so on are identified by their names. In general, code fragments are identified exactly by their textual representation. Copying a text fragments obviously retains this identity. That is to say, source code clones are identical (to begin with). In many modeling environments, on the other hand, model elements have internal identifiers.[4] Since these internal identifiers are usually understood as globally unique, a deep copy of a model element will consistently change the identifiers in the duplicate. Thus, model clones are equal, but not identical.

3.1.4 Names In domain modeling, using just the right names is very important for the effectiveness of the model, and so, renaming may occur frequently, and names may be in a systematic relationship. For instance, if there is a use case "lend medium" in a Library Management System, then

[4] This is the case for most contemporary UML tools, but there are also environments like Adonis [10] where model elements are identified by name.

the activity “lend medium” is probably intended to be a behavioral refinement of the use case, the class “Medium” is likely to play the role in both of them, and the state “lent” of “Medium” would typically be the result of this process. Analyzing the grammatical relationships between these names will require linguistic processing not readily available in many environments.

3.1.5 Concrete Syntax Apart from some experimental programming languages and environments, source code is presented to the user as a string of characters. There are whitespaces and indentations, and editors may add outlining and syntax highlighting, but by and large, source code remains sequential text. Models, on the other hand, have a dual structure. Internally, they are a set of linked meta model class instances. Externally, they are a set of diagrams where secondary notation and particularly layout play a pivotal role in understanding and using diagrams as used in visual modeling languages (cf. [7,37,27]). These two representations may be diverging and modelers may not be equally aware of both structures all of the time.

3.2 Adapting a clone classification

There are several different ways of defining code clones. The following list enumerates the most commonly cited clone type classification (see e. g. [14, 26,39]).

Type I: Exact clone A duplicate that is identical except at most changes to whitespaces and comments.
Type II: Renamed clone A duplicate with consistent changes to identifiers of variables, types, or functions.
Type III: Parameterized clone A duplicate allowing arbitrary changes, additions or removals of parts. This type of clone is also called “near-miss clone”.
Type IV: Semantic clone A duplicate in content only that may be due to code copying, convergent development, or other processes.

This distinction is based both on the algorithms and data structures that are being used to detect clones, and thus to the degree of recall and precision one can expect: [14] lists textual, token, and metric comparison, but also comparison of abstract syntax trees and program dependency graphs. Obviously, rigid token and string comparison will only ever find identical copies, while finding syntactically equal copies requires creating an abstract syntax tree (AST) or parse tree. Finding modified copies requires similarity metrics and heuristics, including tree-similarity measures.

For model clone detection, this classification of clone types and detection methods can not be transferred directly, due to the differences between source code and models we have discussed in the previous Section. For Type I clones, models do not have whitespaces, but they do have secondary notation. Like whitespaces, secondary notation has no influence on the formal semantics, but they are very important for the pargmatic semantics. Similarly, comments are highly relevant modeling concepts: changes to comments should be counted as changes to the model. Finally, in source code, relationships between elements are established by names while the relationships between model elements are established by internal identifiers. An identical deep copy of a model element (i.e., including all its parts) will thus automatically create consistent changes in identifiers, that is, a Type II clone. So model clones of the first type should disregard consistent changes to identifiers.

With regards to Type II clones, observe that names in programs serve two purposes: they are identifiers as discussed above, and they are names with a meaning for humans. In models, however, names only serve the second purpose, and in domain models, bearing the "right" name is the quintessential purpose of many elements. So we interpret this clone type as one with only minor changes to a model fragment, including evolutionary modifications of names, such as correcting a typing error, or adding a word. For instance, an Activity named "lease Book" might copied, renamed to "lease DVD", and slightly changed accordingly. The model clone equivalent of Type III clones, on the other hand, have no such restrictions and afford arbitrary changes. The characterization of Type IV clones can be used for model clones more or less unchanged.

We thus propose the following classification for model clones. In order to highlight the analogy to the established classification yet avoid confusion between code clone types and model clones, we rename the categories from to A–D.

Type A: Exact model clone A duplicate that is identical apart from secondary notation (e. g., layout), and internal identifiers.

Type B: Modified model clone A duplicate with evolutionary changes to the element names, attributes, and parts.

Type C: Renamed model clone A duplicate with substantial changes allowing arbitrary changes, additions or removals of parts.

Type D: Semantic model clone A duplicate in content only that may be due to model fragment copying, methodological or language constraints, convergent development, or other processes.

Orthogonal to the classification in terms of the kind and degree of changes, model clones can also be classified in other ways. *Secondary* clones are pairs of elements e, e' that satisfy the definition of a clone, but are each a part of larger model fragments E and E', respectively, which in turn are clones of each other. For instance, if a class is copied and not changed, all the class properties are secondary clones. Of course, it is preferential to find the *primary* clone rather than a secondary clone (that is, the class), or a (complete) set of secondary clones, but often, finding one element out of a fragment is enough to direct the attention of a human inspector to a certain model fragment and resolve the issue manually.

We will use the term *loophole* clone for duplicates introduced through language loopholes as discussed above in Section 2.3. Finally, for the purposes of this paper, we will also distinguish between *natural* and *seeded* clones, that is, clones that have unintentionally been created by a modeler, and clones that have been planted on purpose.

3.3 Towards a better definition of model clones

As we have argued before, UML models can be regarded as sets of instances of meta model elements which may have links to other such elements. So, a first attempt to define model clones might simply be: any model element e that bears a high degree of similarity to some other model element e'. However, there are many model elements without names or other unique features, that occur in high numbers, and that are necessarily very similar: those duplicates we have called loophole clones (see Section 2.3). For instance, the multiplicities of structural features occur over and over again in class models and are identical up to their object identifier. So, any similarity measure should give a high degree of similarity to them, but intuitively, we would not expect them to be clones. So, detecting true clones will have to avoid showing these to modelers, as these clone candidates will be considered as false positives.

On the other hand, model elements like Class or Activity should not be considered in isolation, as individual model elements. Rather, they are the roots of (containment) trees of elements. For instance, a Class owns a set of Properties and Operations, which may own Types, and Parameters in turn. So, as a second step, we might define model clones as pairs of model *fragments*, where a model fragment is a (complete) tree of model elements. This leads to the following definition.

Definition 1 (Model Fragment, Model Clone) *A* ***model fragment*** *is a set of model elements that is closed under the containment-relationship. A* ***model clone*** *is a pair of model fragments such that there is a a high degree of similarity between the fragments.* □

We will further elaborate and formalize this definition below. Observe, that this definition includes clones of all sizes, from individual elements via larger sets of model elements like a Class to large Packages containing entire subsystems. For an example of how an individual Class really is a set of model elements, consider the Class named "Book" in Fig. 1. Assuming that the two diagram elements named "Class" in the diagram refer to the same model element (with identifier 8) and describe it completely, "Book" has seven parts: the attributes "title", "rec_age", "state", "signature", and "type", two association ends from the associations with identifiers 28 and 92, and the generalization to "Medium". Other typical parts of Classes include Operations, Stereotypes, and possibly even complete behavioral models such as a StateMachine.

Our definition is similar in spirit to the one proposed by Pham et al., who define "*a fragment f is a set of edges of G [a graph* (V, E, T) *of vertices, edges, and a type labeling] which forms a weakly connected subgraph. Two fragments are called a clone pair if they are sufficiently similar with respect to a given similarity measure*" (see [23]). However, this definition ignores the tree-substructures induced by the containment-relationship. This may be a valid assumption for Matlab/Simulink flow models, but is too simplistic for the far more generic and rich modeling language UML, where the many small subtrees are really quite characteristic, as we shall see in Section 4.

While our definition reduces the problems discussed above, it does not completely solve them: Firstly, some loophole duplicates are bigger than some true clones. For instance, ActivityPartitions often contain many more elements than Properties, or even Classes. Secondly, there are still many small similar fragments that would not intuitively be considered clones. It seems, though, that while these are too small to be harmful in practice, their presence considerably hampers automatic clone detection by introducing many false positives (one could call this "noise"), which degrades both performance and detection quality. Thirdly, there are intentional duplicates (cf. Section 2.4) that should not be identified as clones either. But in order to exclude these duplicates from a definition of model clone, we would have to define a model clone as a pair of harmful, unintentional duplicate model fragments exhibiting a large degree of similarity. Obviously, this definition is not helpful in operationalizing clone detection.

The rest of this paper can be seen as testing and validating our definition, determining good measures for similarity that are fast enough to compute, and finding a practical threshold for them.

4 Similarity of model elements

Any attempt to defining clones revolves around similarity, and so we will try to propose useful heuristics for defining similarity. In an approach to systematically derive such heuristics based on facts, we analyze the structure of existing models. To the best of our knowledge, this is a novel approach: no other empirical study of model structures seems to be published.

In the modeling and formal methods communities, it is generally accepted that models (like UML models) are basically graphs, that is, the model information is primarily found in the connections, and all connections are equal. Much of the existing work in these fields incorporates this intuition, including previous approaches to model clone detection (cf. [19, 15, 5, 20], and particularly Pham et al.'s model clone definition in [23] quoted above) which have focused on such kinds of models where this assumption more or less holds, and, consequently, they are using algorithms based on graph structure matching. However, this assumption is simplistic (at least for UML models), as we will show in this section by studying the detail structure of models. This will lead us to four heuristics for model element similarity.

Table 1 Size measures of the sample models used in this Section.

Model	No. of Students	Model Elements	Meta Classes	XMI File [MB]	No. of Diagrams
A	5	6,881	93	2.62	**74**
B	3	4,828	66	1.97	**58**
C	5	5,379	81	3.13	**68**
D	3	2,860	59	1.19	**34**
Total	**16**	**18,893**	115	**9.12**	**234**
Avg.	*4*	*4,723*	78.8	*2.28*	*59*

Table 2 Empirical basis for the concrete measurements in this article. AD: Activity Diagram, AsD: Assembly Diagram ("Composite Structure"), CD: Class D., DpD: Deployment D., IAD: Interaction D., OD : Object D., SMD: State Machine D., UCD: Use Case D.

Model	Number of Diagrams by Type								Total
	AD	CD	UCD	SMD	IAD	OD	AsD	DpD	
A	36	3	27	5	2	0	0	1	**74**
B	33	3	19	1	0	1	0	1	**58**
C	27	3	23	1	7	6	1	0	**68**
D	3	3	26	2	0	0	0	0	**34**
Total	**99**	**12**	**95**	**9**	**9**	**7**	**1**	**2**	**234**
Avg.	*25*	*3*	*24*	*2*	*2*	*2*	*0*	*1*	*59*

4.1 Model Sample

The work reported in this paper is originally based on 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. Instead, we used models created by 16 Master's as part of their assignment in a requirements engineering class taught by the author in 2010. The students worked together in four concurrent groups of three to five participants, each spending around 100 working hours on devising and creating the models. Altogether, every model represents approximately 10 to 12 person weeks of modeling effort. All students followed the same methodology which was taught in that course, and which was almost the same methodology that was used in the industrial projects that lead to our initial observations of the phenomenon of model clones. The students used MagicDraw 16.0, employing a large portion of UML 2.2 (see [21]). Table 2 shows some size measures of these models.

Most of the interaction diagrams were in fact sequence diagrams. The number of model elements is without tool specific elements, profile elements, and diagram data; together, these items typically account for around 80% of the data in a model (i. e., equally the model size in terms of XML nodes and the characters in an XMI file).

Table 3 At first sight, the graph representation of a UML domain model has just the structure intuitively expected from a graph: lightweight nodes, the information is mainly captured by the links.

Model	**Model Elements**	**Element Links**	**avg. degree**	**Element Attributes**	**Attributes per Model Element**
A	6,881	7,844	2.28	6,786	0.99
B	4,828	6,162	2.55	4,975	1.03
C	5,379	6,318	2.35	4,618	0.86
D	2,860	2,072	1.45	1,962	0.69
Total	**19,948**	**22,396**	-	-	-
Avg.	*4,987*	*5,599*	*2.16*	*4,585*	*0.89*
Median	-	-	-	-	-

4.2 Analysis of the model graph structure

Table 3 shows some measurements that we have taken on our samples. First of all, observe that the number of nodes, node attributes, and links are all roughly in the same order of magnitude. If we consider only samples A through C, there are between 12 and 22% more links than there are nodes and between 3% more and 17% less attributes than there are nodes. Also, the average degree of nodes is 2.39, and the average number of attributes per node is 0.96. That is to say, on average in samples A to C, model elements have more than two times as many connections than attributes. Model D deviates a little more with less links and attributes (-38%, and -46%, respectively), and smaller degree and average number of attributes. Also, the model in sample D is much smaller than the other models. All in all, model D contains much less information than samples A to C, but that could well be a consequence of the unfinished state of the model (the students ran into a deadline).

So, at first sight, it would seem that the sample models are indeed graph structured in an intuitive sense: a more or less homogeneous fish-net like structure with rather simple nodes, the semantic information being captured by the links. This would imply that the overall characteristic of a model is dominated by the graph structure, and so, would make a point in favor of using graph-algorithms for matching models and for detecting clones, as previous approaches have in fact done.

However, closer inspection reveals that the vast majority of the links are containment relationships, i. e., they belong to the UML meta attribute ownedMember (85% on average, see Table 4). Only 15% of the links between nodes contribute to a proper graph structure while 85% are spanning many small trees that are mainly used to store semantic attributes of the one node that is the tree root. When adjusting the average degree to cover only non-aggregation links, it becomes clear that the graph structure is much weaker, that is, the relative weight of the node attributes is much higher than that of the non-containment links (approximately 5 times as much).

Table 4 The graph structure of models is mainly an aggregation tree, plus some additional connections.

Model	**Percentage of aggregations**	**True links**	**Number of**			**Tree depth**
			roots	**leaves**	**partitions**	
A	87.7	964	1	4,809	1	11
B	78.3	1335	1	3,162	1	9
C	85.1	940	1	3,882	1	9
D	87.1	268	1	1,269	1	6
Avg.	*84.6*	*877*	*1*	*3281*	*1*	*8.8*

When focusing exclusively on the containment links, on the other hand, we find that in all samples the model is exactly one tree, that is, there is exactly one root and there are no nodes that are not connected to the root. A vast majority of all model elements are leaves (between 65 and 72% for samples A to C, and 63% on average). This fact further confirms our earlier observation that models do not have the graph structure intuitively expected.

To sum it up, UML models are very wide and flat trees. Removing the overall package structure imposed on the sample models by the methodology used, the models are large sets of flat trees. Apart from the containment structure, model elements have few links to other elements, but many attributes. What is more, the majority of the tree nodes cannot exist in isolation (e. g., Multiplicities). That is, they are really just part of a data structure organizing information that belongs to the respective tree's root node, which further increases the relative weight of node attributes over inter-node links as a carrier of semantic information. In other words, UML models are loosely connected fat nodes rather than densely connected graphs of lightweight nodes. The meaning of the sample models is in the nodes more than it is in the graph structure, and there is no reason to assume this observation does not generalize to all UML models. Therefore, we design our similarity heuristics as node similarities rather than graph structure similarities, as previous work has done.

4.3 Similarity heuristics based on element names

Probably the most prominent feature of any model element is its name. Thus, the simplest possible approach for defining similarity among model elements is to use the similarity of their names. This is basically also what a human could do with the model browse and search facilities offered by typical modeling tools today.

Thus we define the first the heuristic NAME as the similarity of the names of two elements, according to the edit distance (aka. Levenshtein distance), or whether adding wildcards at word boundaries or fixing standard typing mistakes (swapped words in sentences, CamelCaps vs. separate

Table 5 Even in domain models, many model elements are not named.

Model	**Model Elements**			**Named instances per meta class**			
	total	**named**	**%**	**min**	**avg**	**median**	**max**
A	6,881	4,961	72.1	1	72,10	26	403
B	4,828	1,364	28.3	1	28,25	27	456
C	5,379	3,351	62.3	1	62,30	12	346
D	2,860	1,741	60.9	1	60,87	11	204
Avg.	*18,893*	*2,854.25*	*55.88*	–	–	–	*352.25*

words) makes the two names match. Model elements that usually do not carry names are excluded from the comparison (i. e., they have similarity 0).

This is justified, because most elements that matter to modelers are named, but not all model elements in a UML model (usually) carry names. In fact, only a little more than half the model elements in our sample are actually named (56% on average, see the first columns of Table 5 for details).

There is a remarkable difference in this aspect between model B and the other models, as model B has less than a third of named elements, while the other models have almost two thirds of named elements. The reason for this variance is that model B has a higher proportion of activity models than the others, and that the activity models in model B are also larger than in the other models. Since activity models often have particularly many unnamed model elements such as ControlFlows and ControlNodes, this skews the overall ratio. In practice, the number of activities relative to other models is typically even higher than in model B; and in the field we have observed up to 95% of all elements in a model being unnamed.

In Table 5, we show the numbers of named instances of meta classes. The average and maximum numbers of named model elements per type are only a small fraction of the total number of model elements. Clearly, comparing sets of less than 30 elements is no computational challenge, and comparing less than 500 elements is still very much feasible even for advanced similarity computations.

Looking at the average, median, and maximum value of the numbers of named instances per meta class in Table 5, it is clear that there are many meta classes with a small population and a few bigger ones. This observation is reinforced by the concept frequency profiles shown in Fig. 8. There, the sample models are presented by the frequency of model elements by meta class normalized to the absolute model size in terms of model elements. We included both named and unnamed model elements.

As one can clearly see, the profiles have both obvious similarities and differences: sample models A and C show very similar profiles, and sample model B has the same spikes, though less markedly. Sample model D, on the other hand, shows a different profile altogether.

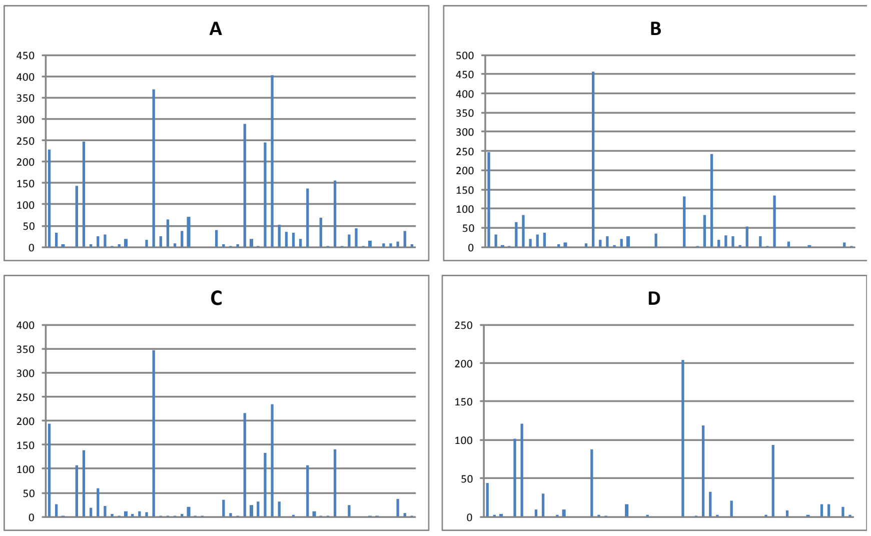


Fig. 8 A rough comparison of the concept frequency profiles of the four sample models: the meta class instances occurring in the model sorted alphabetically (x-axis, same as in Fig. 9) vs. the normalized frequency (y-axis).

In Fig. 9, we show the same concept frequency profiles, but now overlaid and coded with different colors. We have also added the names of the meta classes whose frequencies are recorded. The frequencies have been normalized again to the model sizes. The purpose of this juxtaposition is to show the individual coincidences and differences in the profile, both with respect to meta classes and with respect to the absolute (normalized) magnitude. As one can see, the largest spikes are caused by ControlFlow and ObjectFlow, followed by MemberEnd, AssociationEnd, and Action. The next small group of spikes belongs to Association, Property, and Partition (called ActivityPartition in the UML meta model). Only then come model elements like Class, and Activity; UseCases for instance are so rare in comparison that they do not show up in these measurements. Observe also, that e. g. in sample model A, there are more than five times as many Control- and ObjectFlows than Properties, and again more than five times as many Properties than Classes. This hardly matches the intuition most modelers have about domain models.

This observation also means that a large number of the model elements can be ignored in a name-based similarity. So we can hope to achieve good detection rates quickly even in very large models. The big drawback of the heuristic NAME is, of course, that it is sensitive to renaming—and that is obviously one of the most frequent operations in domain modeling. Also, other attributes of a model element are not taken into account. However, most named elements exist in a context of other elements that are also named. Instead of comparing just the element name, all the names of its neighbors could be considered, too. This leads to the heuristic NAME-2

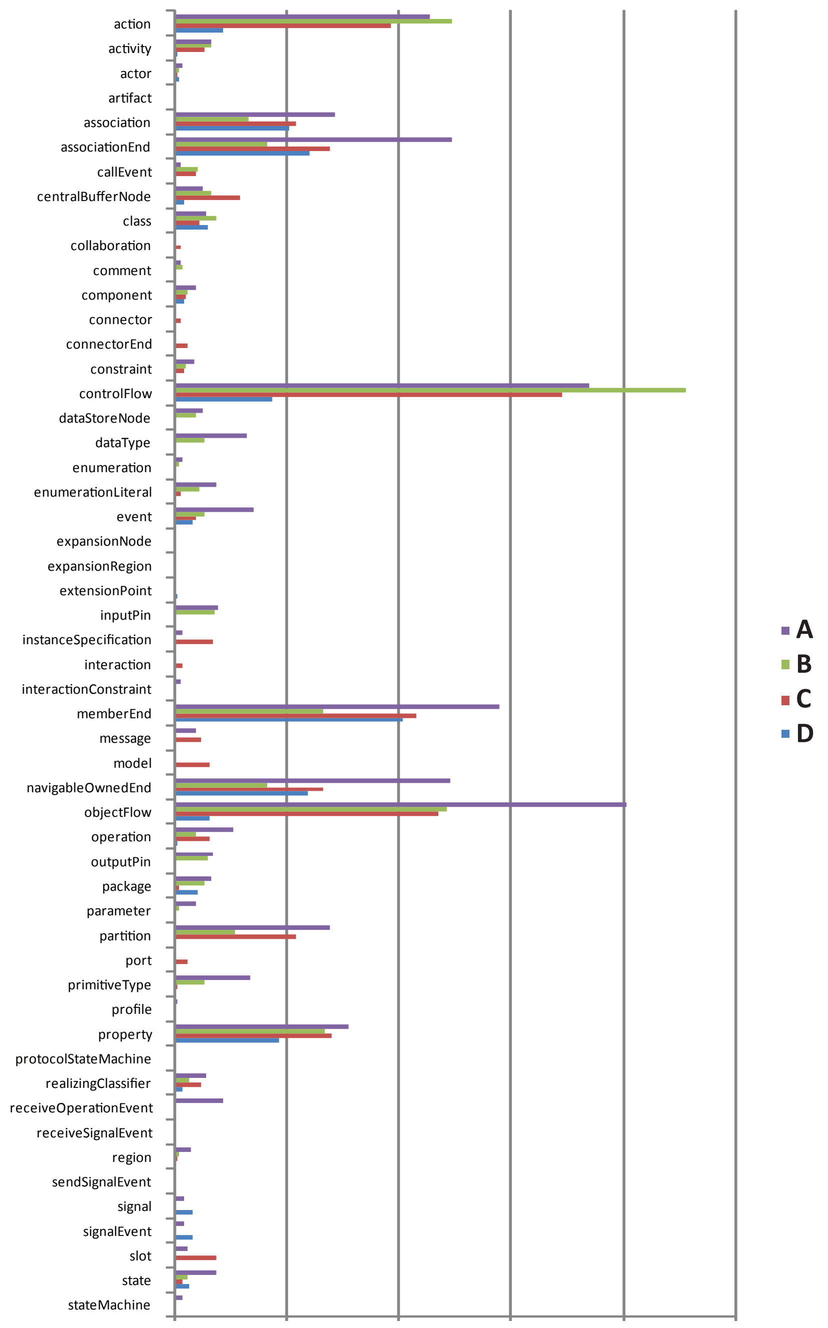


Fig. 9 An overlay of the concept frequency profiles of the four sample models showing the concepts (y-axis) vs. the normalized frequency (y-axis). The overlay clearly shows where the different sample have peaks for the same concepts, and how large the differences are.

that uses not just names, but all kinds of attributes to compare elements, taking the names of linked elements and parts as the values of the link-attributes. We expect this heuristic to perform better than NAME at comparable runtime. Both of these heuristics have the drawback, however, that they put a lot of emphasis on the names of model elements which may work for named elements, but not for others, and it might be too focused on just one attribute. Therefore we also consider a class of heuristics that are somewhat broader, and also have a track record.

4.4 Similarity heuristics based on an element index

One of the most successful family of similarity heuristics known from code clone analysis is based on indexing functions.[5] This approach has demonstrated several attractive features for code clone detection: it yields good detection rates, it is equally suitable for all clone types, and it is yet one of the fastest approaches. So it might be a good heuristic in the detection of model clones, too.

A function h computes suitable indices of two model elements e_1 and e_2 which can then be compared yielding the similarity. We define the similarity heuristic INDEX by $\frac{|h(e_1)-h(e_2)|}{h(e_1)+h(e_2)}$, where h is realized as simply adding up all the characters of the representations of e_1 and e_2, respectively. All links are included in this computation, i. e., if the representation includes references to connected and contained elements, the respective element identifiers are simply taken as character strings.

Frequently, there are many such references, and thus, it is critical how long these identifiers are. For instance, the internal identifiers used in MagicDraw, for instance, consist of 40 characters, 8 of which identify the tool version. In our implementation, we use a very compact representation of models (see Section 5 below for details), where internal identifiers are usually small integers, i. e., they are typically between one and four characters long. Table 6 shows some measurements on our samples, relating the numbers of model elements and identifiers, and the number of characters used in identifiers, names, and the complete model. In our sample, around 8% of all characters of a file belong to an identifier. In contrast, only just over 2% of all characters are used for the names of model elements.

So, if long internal identifiers are used, "identical" copies of fragments may differ in up to 8% of the characters used to represent the copies. Thus, an index-based similarity function might not work well for XMI models, but it might work a lot better if shorter identifiers are used.

However, if internal identifiers should wreck index-based similarity, it might help to replace internal identifiers with the names of the elements re-

[5] This approach is often called hash-function-based, but that name is misleading, of course, for the objective of a hash function is to distribute all inputs evenly over the key set while for clone detection, it is crucial that $|h(e_1) - h(e_2)|$ is more or less proportional to the similarity of e_1 and e_2.

Table 6 Number of identifiers and names, absolute numbers of characters in identifiers and names, and percentages of identifier and name characters as compared to overall file characters. ME abbreviates Model Elements.

Model	Number of Identifiers	Number of ME	Characters in IDs	[%]	Characters in Names	[%]
A	14,725	6.881	58,9000	7.8	14,796	2.51
B	10,990	4.828	43,9600	7.7	11,356	2.58
C	11,697	5.379	46,7880	9.3	9,056	1.94
D	3,877	2.860	15,5080	7.2	3,489	2.25
Avg.	10,322.3	4.987	41,289	8.0	9,674.3	2.32

ferred to. So we define the heuristic INDEX-2 as INDEX before, but before h is applied, all identifiers are replaced by the names of the elements their refer to (or their type, should there be no name). This similarity function should perform better than INDEX, both on models with long and short identifiers. At the same time, it should also be less sensitive to renaming than NAME.

5 Detecting model clones

5.1 Formal structures of models

We have concluded Section 4.2 with the observation that UML models are not densely connected graphs of lightweight nodes, but rather loosely connected graphs of heavy nodes, and we have used this observation for proposing similarity heuristics based on nodes rather than on networks.

However, models can still be considered as graphs in the mathematical sense. In order to arrive at a precise and operational definition of model clone, we now use this fact to formally define models, model fragments, and thus, model clones. We start with defining a special kind of labeled graph suitable for representing models.

Definition 2 (Model Graph) *A* ***Model Graph*** *is a 10-tuple of the form* $\langle N, \mathcal{T}, type, E, source, link, \mathcal{A}, slot, \mathcal{V}, val\rangle$ *such that*

- *N and E are finite sets of nodes and edges, respectively, with $E \cap N = \emptyset$;*
- *$\mathcal{T}$ is a domain of types representing the modeling concepts (i.e., the meta classes of the modeling language);*
- *$type : N \to \mathcal{T}$ is a function equipping every node with a type;*
- *two functions $source : E \to N$ and $link : E \to N$ defining the origin and target of each edge;*
- *$\mathcal{A}$ is a domain of attribute names representing the modeling concepts' properties (i.e., the meta attributes of the modeling language);*
- *$slot : N \to 2^{\mathcal{A}}$ is a function associating every node with a set of attribute names; the same name is used to associate attribute names to edges, i. e. $slot : E \to \mathcal{A}$;*

- *$\mathcal{V}$ is a domain of values representing the properties' values (i.e., the values stored in the slots realizing meta attributes); and*
- *$val : N \times A \rightharpoonup \mathcal{V}$ is a partial function associating a value to every combination of nodes n and attributes a defined on that node, i.e., $a \in slot(n)$.*

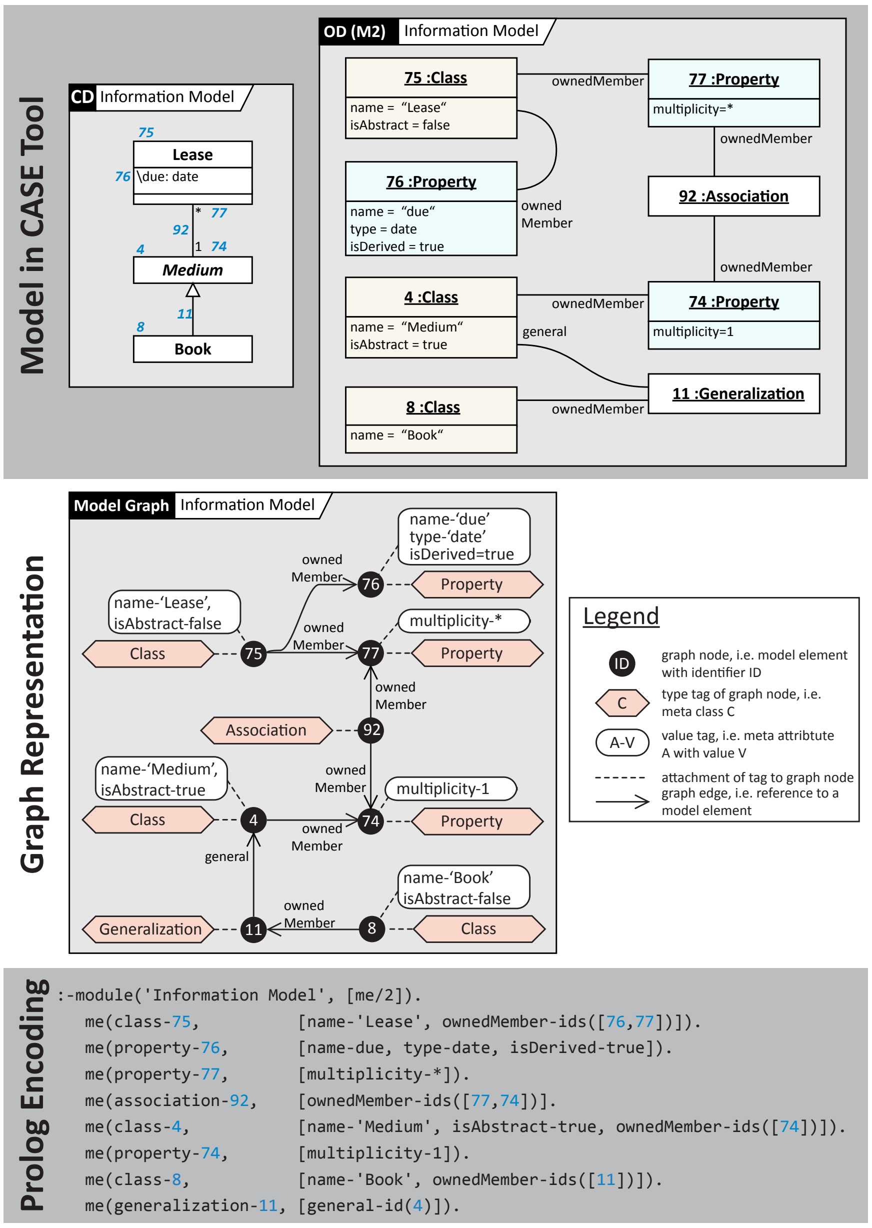
The notation g_x is used to access the component x of a graph g, i.e., g_E denotes the edges of g. It is also required that $\forall e, e' \in E : (slot(e) = slot(e') \wedge source(e) = source(e')) \implies e = e'$ for all model graphs. □

Model graphs may represent both complete models and model fragments. The names *slot* and *link* in this definition are motivated by the corresponding UML terminology. The codomain of *link* is not sufficient to completely represent relationships whose target is an ordered set of elements, but using $link : E \to N^*$ instead would make the definitions overly complex without significant benefit.[6] $\mathcal{T}$, $\mathcal{A}$, and $\mathcal{V}$ are determined by the underlying meta model and are included to make the definition self-contained so as to be applicable to instances of arbitrary meta models, e. g., different versions of UML and, eventually, languages other than UML.

Fig. 10 shows visually, how a model fragment is translated (we will refer to the first two bars only for the time being). The first step is not a translation as such, but simply a different view on the same entity: inside any UML-compliant tool, the model query in Fig. 10 (top left) is already represented in a data structure very similar to the structure outlined by the object diagram shown in Fig. 10 (top right). Such an object structure directly corresponds to a graph in the mathematical sense, as shown in Fig. 10 (middle). Here, we depict nodes as black filled circles with white numbers denoting their identity. Nodes are always tagged by a type, shown in hexagons. Nodes may have any number of slots represented as rectangles with rounded corners attached to them which show the current value and the name of the corresponding attribute or operation (a Feature, in UML terminology). Nodes may also be connected by labeled edges which we represent as directed arcs. Observe that the layout and the identifiers used in Fig. 10 are consistent to support tracking the elements. We can now elaborate and make more precise our definition of model clones (Definition 1, p. 12).

Definition 3 (Model Fragment) *Given a model graph M, a* ***Model Fragment*** *F of M is a sub-graph of M, i. e., $N_F \subseteq N_M$, $E_F \subseteq N_M$,*

[6] There are in fact many occurrences of meta attributes and associations with multiplicity greater than 1 and constraint `ordered` in the UML meta model. However, in the samples we have studied in the research leading to this article, the only UML meta attributes that ever needs a multiplicity greater than 1 is ownedMember, or one of its specializations like Association.ownedEnd, Interface.ownedAttribute, or Operation.ownedParameter. For these link types, ordering is irrelevant.



```
:-module('Information Model', [me/2]).
   me(class-75,            [name-'Lease', ownedMember-ids([76,77])]).
   me(property-76,         [name-due, type-date, isDerived-true]).
   me(property-77,         [multiplicity-*]).
   me(association-92,      [ownedMember-ids([77,74])]).
   me(class-4,             [name-'Medium', isAbstract-true, ownedMember-ids([74])]).
   me(property-74,         [multiplicity-1]).
   me(class-8,             [name-'Book', ownedMember-ids([11])]).
   me(generalization-11, [general-id(4)]).
```

Fig. 10 The transformation of a model (top left), its tool-internal data structure representation as defined by the UML standard (top right), a conceptual representation as a labeled graph (middle), and finally the resulting Prolog code (bottom). Note that layout and identifiers are consistent to support tracing. The Prolog code has been simplified by removing tool-specific information like version, loaded libraries and stereotypes, and information about the model (UML version, view type, and so on). The identifiers are the same as in Fig. 1.

and so on, and F is a model graph in itself according to Definition 1. We also require that

$$link_M[\{e \in E_F \mid slot(e) = \texttt{ownedMember}\}] \subseteq E_F,$$

i. e., F is closed under the containment relationship. We use the abbreviation $f[X]$ to stand for $\{f(x) \mid x \in X\}$. □

As we have explained in Section 2 above, there is an element of subjective interpretation in determining which duplicate fragments actually are or are not clones. For a human clone detector, it is most important to get most of the true positives first without a high number of false positives. Since this is a situation quite comparable to web search, we define a clone only relative to a threshold, thus yielding a ranking of clone candidates.

Definition 4 (Binding, Similarity, t-clone) *Given a model graph M, a threshold $t \in [0..1]$, and two model fragments o and c of M representing an original and a clone. A **Binding** β between o and c is a bijective relation $\beta \subseteq o_N \times c_N$ among the nodes in the fragments o and c. We say the **Similarity** sim of a binding β is*

$$sim(\beta) = \sum_{\langle e_1, e_2 \rangle \in \beta} \sigma(e_1, e_2)$$

*for an element-based similarity function σ, that is, the sum of all the element similarities in a fragment. We say that c is a **t-clone** of o, if and only if β is a binding between c and o and $\sigma_\beta \geq t$.* □

Definitions 4 supersedes Definition 1.

5.2 A clone detection algorithm

Based on the definition of model graph, we may now define a clone detection algorithm (see Algorithm 1 below). Observe that it is parametric in the definition of similarity between two model elements: any of the four various similarity heuristics defined above may be inserted for σ. This way, it is easier to study and compare the alternative notions of similarity on our way towards an optimal clone detection algorithm.

5.3 Implementation

In order to evaluate the algorithm and the heuristics, we have implemented the $\mathsf{MQ_{lone}}$ tool (pronounced "m clone"). It is based on earlier work on model matching and querying models (see [30,32,33,38,36]), and reuses parts of our MQ tool for model querying, which also explains the name. $\mathsf{MQ_{lone}}$ transforms XMI files from many contemporary UML CASE tools like MagicDraw, transforms them into Prolog programs as described in Fig. 10,

Algorithm 1: Clone detection algorithm outline

```
Input: a model graph $M$, a threshold $t$, a set $RC$ of relevant concepts
Output: a partial function from pairs of nodes of M to similarity values

find:
  $Candidates \leftarrow \{C_t \mid t \in RC\}$, where
  $C_t = \{\langle e_1, e_2\rangle \mid e_1 \neq e_2 \wedge type(e_1) = t = type(e_2)\}$;

compare:
  $Result \leftarrow \emptyset$;
  while $C_t \in Candidates$ do
    forall $e \in C_t$ do
      $E_e \leftarrow$ transitive closure of $e$ under ownedMember;
    end
    forall $e_1 \in C_t, e_2 \in C_t, e_1 \neq e_2$ do
      $Bindings \leftarrow$ all bindings between $E_{e_1}$ and $E_{e_2}$;
      $s_{max} \leftarrow 0$;
      forall $\beta \in Bindings$ do
        $s_{max} \leftarrow max\{s_{max}, sim(\beta)\}$;
      end
      $Result \leftarrow Result \cup \langle e_1, e_2, s_{max}\rangle$;
    end
    $Candidates \leftarrow Candidates - C_t$;
  end

select:
  $Result \leftarrow \{\langle e_1, e_2, s\rangle \in Result \mid s \geq t\}$;
  sort $Result$ by decreasing $s$;

return $Result$;
```

detects clones in them, and presents the clone candidates back to the user. MQ$_{\mathsf{lone}}$ consists of a plug-in to the MagicDraw UML CASE tool written in Java and an algorithmic core implemented in SWI Prolog [41].

We look at models as knowledge bases, and we represent individual model elements as individual facts in a knowledge base implemented in Prolog. There are several advantages of using Prolog to encode models and model operations. First, any Prolog system is already a powerful and easy to use repository, allowing for a tight integration of data and functions. This, in turn, gives rise to rather efficient implementations and avoids a separate data base (or re-parsing model files every time). Second, the declarative programming style is very natural for expressing rules describing the similarity or difference of model elements. Third, the tight integration between data and function, the high level of abstraction of Prolog, and the interactive execution make it very easy to quickly change the system. Thus, we may experiment with different settings, rules, and algorithms—which is essential to us, since this whole effort is explorative by nature, and interactive operation and rapid turnaround are key requirements.

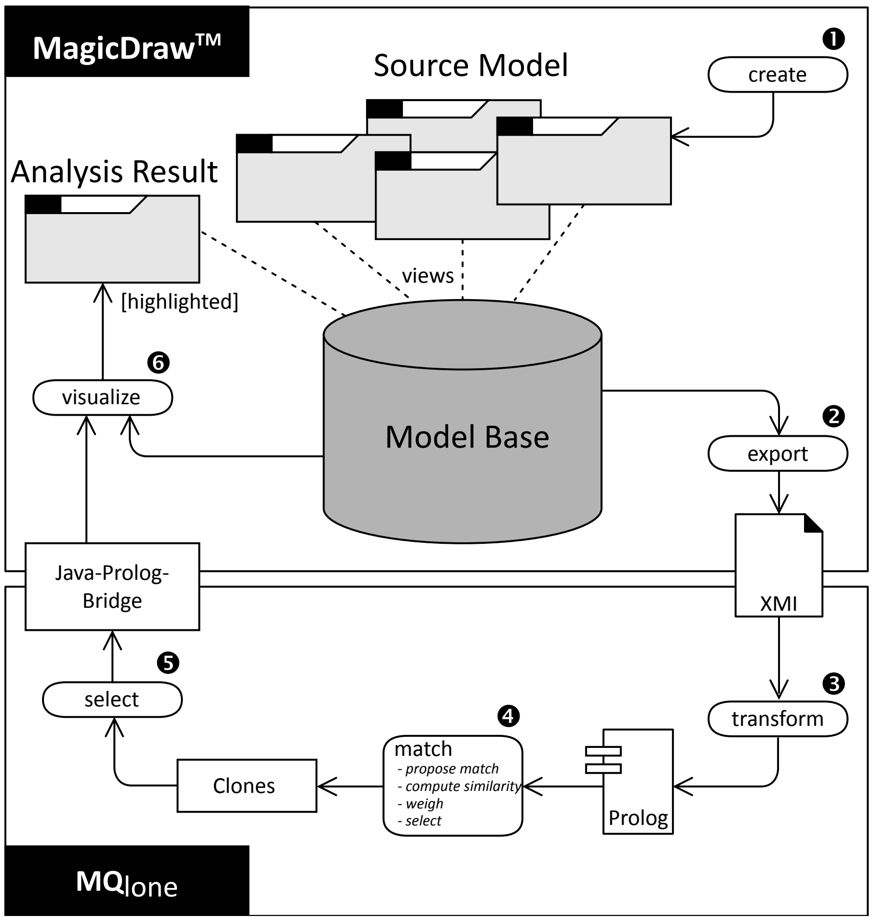


Fig. 11 Implementation of MQ$_{\text{lone}}$; the numbers in black circles refer to steps in the main data flow, see Section 5.3 for more details.

The transform of UML models into Prolog fact bases is a very straightforward process, see Fig. 10 (bottom): model graph nodes become facts of the `me` predicate, and model graph edges are represented by references to other nodes. If there are several edges in a model graph that start from the same node and that carry the same slot name, they are joined into one attribute in the implementation (see the list-valued `ownedMember` slot of `class-1` in see Fig. 10, bottom).

The actual implementation of matching and similarity deviates a little from the specification shown in Algorithm 1, mainly for optimization purposes, e. g., for exploiting Prolog's extremely fast pattern matching capabilities to avoid enumerating pairs of elements with different and/or unsuitable types. Further optimizations like ordering the constraints reduce the search space further.

Fig. 11 presents an overview over the $\mathsf{MQ}_{\mathsf{lone}}$ system and the data and control flow for clone detection. The main steps are labeled by numbers in black circles.

❶ First, a model is created manually or by generation or transformation.
❷ Second, the model is exported to the standardized XMI file format using the built-in facilities of the MagicDraw UML CASE tools.
❸ The XMI file is transformed into Prolog code. Note, that this is a purely syntactical transformation. No information is added or removed.
❹ Next, the clone detection as such happens. It is based on model matching and similarity, as described above. The result is a prioritized list of pairs of model elements together with their similarity, and the reliability of this result.
❺ Then, the most promising clone candidates are selected, e. g., taking a quantile or a fixed number of candidates.
❻ Finally, the result is visualized as a table.

A more convenient visualization is currently being developed.

6 Evaluation

6.1 Method

In order to evaluate the quality of our approach, we ran our implementation on one of our sample models of the Library Management System (LMS). First, we selected the sample model that was the biggest and had received the best grade (model A in the following tables), expecting it to have the smallest number of natural clones. Indeed, a manual inspection of the remaining model did not find any natural model clones. Then we manually seeded the sample model 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), changed them to emulate Type A, B, and C model clones, and marked both the nine original model fragments and the nine copied (and modified) model fragments manually as originals and clones, respectively. We did this by attaching comments to the elements, that is, the elements as such were not changed since the connection between an element and its comments is established by a link in the comment.

This resulted in 145 model elements being marked as clones and 155 being marked as originals, out of a total of 7181 model elements in the model after seeding, i. e., approx. 4.2% of the model elements were marked. The annotation allowed automatic computation of precision and recall. Table 7 shows the result of the seeding process. The sizes of the clones are given as the number of model elements in the duplicate fragment (i.e., nodes of the containment tree owned by the main element), and the number of filled

Table 7 Manual seeding of nine duplicate fragments resulted in 145 model elements being marked as clones. ME standas for Model elements, A stands for number of element attributes.

Clone	**Size (ME/A)**			**Changes**	
Type	**UseCase**	**Class**	**Activity**	**Name**	**Features**
A	7/4	9/68	41/239	no changes	no changes
B	3/12	8/68	22/125	small changes	small changes
C	1/6	8/72	46/259	different name	add/delete
Sum	11/22	25/208	109/623		

meta attributes in the fragment. The last two columns show what changes were applied to copies.

Initially, we ran the clone detection algorithm with a selection quantile of 50%, thus yielding a mixture of seeded and natural clone candidates. In order to assess the natural clone candidates, we manually reviewed them, and annotated those as "natural" that qualified as clones according to our definition; almost all of them were loophole clones.

6.2 Measurements

As we have explained in Section 3 above, there is an element of subjective interpretation in determining which duplicate fragments actually are or are not clones. Manually following up on a clone candidate proposed by a tool is an expensive process. So, from a practical point of view, a modeler is not so much interested in eventually obtaining the complete list of all clones. It is much more helpful to quickly get a short list of very good matches. Then the modeler can assess them, and repeat the process until he feels satisfied with the quality, or running out of resources. So, the objective with the algorithm is not to maximize true positives (i. e., recall), but to minimize false positive (i. e., precision). Since the sample model very likely did not contain natural clones prior to seeding, we compute recall and precision only based on the seeded clones.

We repeated the clone detection, varying the heuristics used, and recording the detection rates. The measurements were conducted on a subnotebook (Centrino Duo, 1.2GHz, 2GB RAM). The results are shown in Table 8.

The four blocks of Table 8 show results for the four heuristics NAME, INDEX, INDEX-2, and NAME-2. Each of them is presented for the first 10, 20, and 30 results. For each of these treatments, the first block of lines shows the precision and recall rates as percentages, and then false and true positives. A false positive is an element detected as a clone which is not actually a clone. A Loophole clone is a clone that is generated through a language loophole as discussed in Section 2.3.

The next block of lines deals with the different kinds of seeded clones. Recall that we seeded three groups of clones with an increasing amount of changes, where Type A clones are identical to the originals (modulo internal

Table 8 Measurements of clone detection quality: the number of results is set by the user; black and white boxes in rows Type A-C clone indicate detected and missed seeded clones, respectively.

Heuristic	**NAME**			**NAME-2**		
Results	**10**	**20**	**30**	**10**	**20**	**30**
Precision	60%	50%	67%	100%	80%	67%
Recall	0%	0%	0%	67%	67%	67%
False Positive	4	10	10	0	4	10
True Positives	6	10	20	10	16	20
Loophole Clones	6	10	20	5	9	10
Type A Clones	□□□	□□□	□□□	■■□	■■■	■■■
Type B Clones	□□□	□□□	□□□	■■□	■■■	■■■
Type C Clones	□□□	□□□	□□□	□□□	□□□	□□□
Run Time	5.4s	5.7s	5.5s	5.7s	5.6s	5.7s

Heuristic	**INDEX**			**INDEX-2**		
Results	**10**	**20**	**30**	**10**	**20**	**30**
Precision	50%	40%	33%	80%	55%	50%
Recall	44%	56%	56%	44%	56%	56%
False Positive	5	12	20	2	9	15
True Positives	5	8	10	8	11	15
Loophole Clones	0	1	3	1	4	5
Type A Clones	□■■	□■■	□■■	□■■	□■■	□■■
Type B Clones	□■■	□■■	□■■	□■■	□■■	□■■
Type C Clones	□□□	□■□	□■□	□□□	□■□	□■□
Run Time	5.5s	5.5s	5.7s	5.6s	5.7s	5.7s

identifiers), Type B clones have been somewhat changed, including the name and structure, and Type C clones have been substantially changed including replacing the name, and deleting/adding substantial parts of the structure. In each group, we seeded a UseCase, a Class, and an Activity with all their transitive parts. Detected and missed seeded clone are shown as black and white boxes in the table, respectively. So, for instance, □■□ means that both the seeded UseCase and Activity clones were not detected, but the seeded Class clone was correctly identified.

The last line presents the run time. We here show the average of three subsequent runs to cancel out any effects due to garbage collection and similar factors.

6.3 Detection quality

The heuristic NAME (top left) has an acceptable precision, but a poor recall: it fails to discover *any* of the seeded clones. All the clones it does discover are loophole clones, i.e., they are implied by the modeling language. For all practical purposes, thus, this heuristic is useless.

The heuristic INDEX (bottom left) performs worse in terms of precision, but detects a fair number of seeded clones (recall of seeded clones is around 50%). It detects only few loophole clones ($\leq$ 10%), and has only slightly elevated numbers of false positives among the first 20 hits as compared to the heuristic NAME.

The heuristic INDEX-2 (bottom right) exhibits the same recall, but an increased precision as compared to INDEX. It detects less false positives (in particular when looking at the first 10 matches), but finds slightly elevated numbers of loophole clones.

The heuristic NAME-2 (top right) performs much better than all the other heuristics in terms of false positives and precision. It also outperforms the other heuristics in terms of recall. The number of detected loophole clones is larger than for the INDEX heuristic, but slightly smaller than that of the NAME heuristic.

As expected, Type C clones are less often discovered than Type A and B clones, although the heuristic INDEX discovers one Type C clone, while the heuristic NAME-2 does not. Also, small clones are less often discovered than larger clones: recall that the triplets of black and white boxes represent UseCase, Class, and Activity clones, in that order, and we have seen before, that the instances of these three meta classes have a similar ordering with respect to their average size (see again Table 7). Thus it is no surprise, that INDEX and INDEX-2 discover none of the UseCase clones, but they do discover a Type C Class clone. The Type C Activity clones, on the other hand, are not discovered. We attribute this *also* to the large size of Activities: Since the element indices simply add up, differences are bound to cancel each other out with increasing numbers of elements. Thus, index based similarities turn into size metrics for large model fragments. It is also remarkable, that NAME-2 does discovers both Type A and Type B UseCase clones among the first ten hits, but not Type A and Type B Activity clones. We attribute this, again, to the larger size of Activities as compared to Classes and UseCases: the relative weight of names decreases with an increasing number of other, unnamed elements.

Our expectation that INDEX-2 would perform better than INDEX was confirmed, but to a much smaller extent than we expected. The hypothesis that index based similarities would be less sensitive to renaming than NAME-2 was indeed confirmed: both INDEX and INDEX-2 detected some Type C clones (i. e., duplicates with new names), while NAME-2 did not detect these clones. Conversely, the index based similarities did not detect the smallest clones while NAME-2 did. This is due to the large number of small elements that are rather similar to these small clones.

6.4 Detection runtime

In source code clone detection, runtime is a crucial aspect of the usefulness of an algorithm. Since models, too, can reach quite substantial sizes in

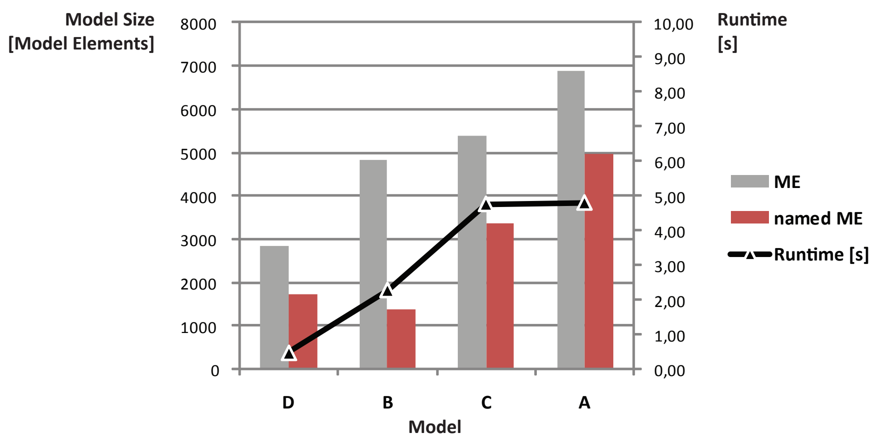


Fig. 12 Runtime measurements for the heuristic NAME-2 on unseeded models (cf. Section 4).

practice, we have also studied the runtime of our algorithm. In the detection quality experiment described in the previous section, the heuristic NAME-2 performed best in terms of quality, while being only slightly worse in terms of runtime. Thus, we focused on the heuristic NAME-2 and disregard the runtime behavior of the other heuristics.

We have measured the detection duration of our algorithm on the original models studied above (see Section 4). Since they have both different sizes and structures, they can both inform us about the relationship between size and runtime, and help assess the influence of model structure on the runtime. Fig. 12 summarizes our measurements.

Observe that the time measured for model A is smaller than the time measured in the previous experiment (see Table 8). This is because the seeded version of model A is about 25% larger than the unseeded version which we used here.

6.5 Interpretation of findings

We have studied four sample UML domain models and found some surprising results. While the representation of models we use might be understood as a graph in the mathematical sense, thus putting our approach in the same box as graph based approaches of the past, such an interpretation does not adequately reflect the vast number of types of nodes, and the large number of complex attributes that they may have. According to our sample, models are not graphs with lightweight nodes and a dense connection structure where the model information is mainly in the edges. Rather, UML domain models can be seen as sets of heavy nodes that carry the majority of the information, plus a few links. Also, many model elements are not named, and tool-specific details of the internal representation (e. g., of identifiers) influence the suitability of clone detection algorithms significantly.

As we have outlined before, the simplest possible approach is to look for name similarities, like we did with the heuristics NAME. This is also what a human could do with search facilities offered by typical modeling tools, so this heuristic also serves as a kind of null hypothesis. Clearly, all other heuristics perform significantly better than this, so we can claim that the approach offered by this article is better than the naive approach to clone detection. It is also much more cost-effective than manual inspection, and it is easily repeatable as a regression check after model evolution. As Table 8 shows, elaborating on the similarity heuristics to encompass more information for each comparison seems to improve the detection quality (compare NAME with NAME-2, and INDEX with INDEX-2, respectively).

Our expectation that INDEX-2 would perform better than INDEX was confirmed, but to a much smaller extent. However, the expectation that INDEX-2 would perform better than NAME-2 was not confirmed. Using shorter strings as identifiers considerably reduces the noise of an index-based clone detection algorithm, but generally speaking, index-based clone detection does not perform as well for model clones as it is reported to do for source code clones. Probably the most remarkable observation is the detrimental effect the model fragment size seems to have on the detection quality. This definitely requires further investigation.

All the heuristics proposed are fast enough for practical purposes on medium sized models. The runtime seems to grow approximately at the same rate as the input (roughly 1s per 1000 model elements), but more experiments with larger models and models developed with different methodologies will be needed to derive more reliable conclusions.

6.6 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. [26, p. 87]). However, there have been some practical studies on the role of code and model clones, and also this paper is founded on studying a set of realistic models. But there is definitely a need for establishing beyond doubt that model clones exist as a substantial modeling problem in practice. This could be achieved only through large scale field studies.

The generalizability of the findings reported in this paper is limited by the number and the nature of the models used to develop and validate our approach. The model sample was not created in an industrial but in an academic environment. However, judging by personal experience, students are quite comparable to domain modelers in industry with respect to the levels of expertise and motivation. Also, the work reported in this paper is originally based on the author's experience from two very large scale industrial modeling projects.[7] That is to say, while we cannot offer evidence from proper case studies to support our claims, we have at least anecdotal evidence from massive industrial projects (see [34] with a few details on one of them).

Also, the sample may be considered too small and too homogeneous: we use only four models, and they are all created with the same methodology, thus exhibiting structural similarities. Larger sample bases will have to be studied before we can be more confident about the applicability in other contexts than MSc-level education.[8] Obviously, it is very difficult to get access to industrial models, a problem that has hampered progress in the field for a long time.

Finally, one might object to seeding clones manually as we have done. Our main objective was to introduce a wide variety of clones rather than introducing a realistic and representative profile of clones: as we have said before, there are no published results available that would inform us on such profiles. Since the primary purpose of the work reported in this article is to develop algorithms, however, we think manual seeding with constant frequencies across all categories is not just acceptable, but actually essential.

For future work involving "natural" clones, keep in mind that despite our present efforts, there is no universally accepted operational definition of code clone, let alone model clone, and ultimately, it is up to human judgment what is and is not a (model) clone. Thus, manually finding a comprehensive and universally accepted list of all clones in a model of realistic dimensions is not just excessively costly and time consuming: it is effectively impossible to do for arbitrary models.

7 Related work

The core concept underlying our approach is that of matching models, model fragments, and model elements. In version control, matching is typically computed between two subsequent versions of the same model. So, one can expect the two models to be roughly the same size. Also, they are usually created using the same model and share the vast majority of the model element identifiers. That means, if two elements from the different models have the same identifier, they can be considered to be identical, even if their meta class has changed (e. g., refining *Action* to *CallActivityAction*,

[7] Due to legal and technical constraints, however, we could not use the models from these case studies directly for this paper.

[8] Observe, however, that most existing work in this area lacks empirical validation altogether; those that do report on empirical results at all employ similar sample sizes (e. g., [9] have $n = 5$).

or *Class* to *Component*). That means that the matching is extremely easy, and so, similarity computation can focus on individual model elements, a very cheap process. On the other hand, the use case version control needs very fast matching computations to be practical.

In model querying, matching is typically computed between one large model (the model base) and one very small model (the query). Model identifiers can not be used for matching, of course, but by cleverly choosing the first element of our query submodel which we try to match with the model base will speed up the process enormously. For ad-hoc querying (that is, an interactive process), all that really matters is the latency between delivering two subsequent query results. Since a user is in this loop, response times in the low second range are acceptable. Also, recall is not really an issue, as long as the high-precision results are presented to the user first. When using queries as heads of transformations, however, the overall duration for computing *all* matches is relevant, and too big recall may trigger transformation rules too often, with very detrimental effects.

In model clone detection, matching has rather different constraints, though. If a clone is potentially any submodel, theoretically, all of exponentially many submodels of a model have to be matched with the whole model. Also, identifiers can not be used to support matching, for even a value copy of a model element will receive a new object identifier. So, already matching of elements needs to compute a similarity measure between them. Obviously, this is a very expensive strategy. It is not clear which of the several classes of algorithms known for code clone detection might be the best for model clone detection of UML models.

As we have mentioned before, there is a large body of work on code clones: [14] provides a survey of the field, and [3] gives a very recent overview of the state of the art. Clones in models, in contrast, have received much less attention. Only in the last few years have there been investigations into this topic. They can broadly be divided into four classes. First, there are approaches to find clones in Matlab/Simulink flow graph models: CloneDetective [6,5] and ModelCD [23,20], both of which use graph isomorphy algorithms. This is adequate for Matlab/Simulink models, but not for the much richer and much more diverse model structures found in UML.

Second, there are approaches that explore model matching for version control of models. Alanen and Porres [1] study set-theory-inspired operators on models. Kolovos et al. on the other hand have proposed the Epsilon Merge Language (EML, [13]) using a identifier-based matching process, while Kelter et al. [12] uses the Similarity Flooding algorithm in their SiDiff tool. All of these approaches, however, are very much locked onto the UML (in particular, class models) and its technical infrastructure (i. e., MOF/XMI), which restricts its applicability. Also, in version control one can reasonably expect most model elements to have the same unchanged internal identifier in two versions that are to be merged. Thus it is easy to find a high-quality mapping to seed a matching algorithm. In clone detection, however, the problem is to find the mapping in the first place.

Many graph- and tree-matching algorithms (see [24] for a survey). For instance, Similarity Flooding [16] is a fixed point computation that may take many iterations. Given the large number and size of potential mappings between duplicate fragments, it is unlikely that such algorithms will be applicable to clone detection. 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. [17, p. 3]). The heuristics we propose, however, appear to scale linearly.

Third, there have been approaches that have explored graphs and graph grammars as the underlying data structure of all types of models (cf. the PROGRESS system, [18,28]). These approaches have developed graph matching algorithms that could be used for clone detection, but, to our best knowledge, have not been studied under this angle and are currently not pursued any more. Also, as we have clearly shown in Section 4, UML domain models possess a structure that is a graph technically, but bears little similarity with the common intuition of a graph. Rather than relatively dense and homogeneous networks of light weight nodes, UML domain 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.

Fourth, there are various approaches dealing with matching of individual UML model types such as interactions [15,25] or state charts [19]. In contrast to the approaches mentioned above, this article deals with all types of UML models, including class models and flow-like models such as activities.

The most important difference between our approach and the related work, however, is the fact that our approach is targeted to and works on domain models, that is, artifacts created in the analysis or requirements elicitation phase. In contrast, all the approaches mentioned above work on design or implementation models, that is, models that are oriented to the underlying target architecture and technology, or that *are* the implementation (in the case of the Matlab/Simulink models). While we can not prove it at this point, personal experience suggests that the structure and characteristics of domain models differ greatly from models at later stages in the development process.

8 Discussion

8.1 Summary

Code clones are a substantial problem for code based development, and model clones are increasingly becoming a problem for model based development. However, currently, there is not much published work on model clones, and next to no work on UML model clones.[9] Therefore, this article started out analyzing actual model clones in UML domain models, and

[9] An earlier stage of the work reported in this article has been published as [35], but the present article has little in common apart from the basic ideas.

proposed a terminological framework, a pragmatic definition, and a clone classification schema adapted from work on source code clones. We developed a clone detection algorithm and model element similarity heuristics based on a detailed examination of actual model structures. Given that many algorithms are proposed based on intuitions or analytical arguments alone, we think this is a particularly sound approach that increases the validity of our approach. We also provided formal definitions of models, model fragments, and model clones, and implemented our approach in the $\mathsf{MQ}_{\mathsf{lone}}$ tool (pronounce as "m clone"). The detection quality and runtime of our algorithm and heuristics were validated experimentally. Finally, we surveyed the related work, and discussed our observations.

8.2 Scope and limitations of approach

All models used in this study are UML 2.2 domain models that were developed using the MagicDraw modeling tool and applying the same methodology as part of student's classwork. The approach applies to *all* kinds of UML models while similar approaches cover only a single model type (e. g., UML State machines, or Matlab/Simulink models). Our approach is likely to generalize to (with decreasing confidence): other versions of UML, other modeling levels (i.e., design and implementation models), other modeling methodologies, and models of industrial origin. Probably the greatest limitation to our work is the lack of validation with truly representative clones and truly representative models. Such models, however, are extremely difficult to obtain with the intent to publish.

Our approach may not work as well for MOF-based modeling languages other than UML, and very likely, it will be unusable for modeling languages from completely different conceptual realms (such as ARIS, ADONIS, or IDEF), since the MOF/XMI representation is exploited in many places. The approach works well for small and medium sized models, but no tests have been run on large and very large models, partly since such models are difficult to obtain, partly because as of now, our implementation is only a prototype and so any performance results would not be very reliable and conclusive anyway.

This work was originally intended only as a research vehicle allowing us to experiment with different algorithms and parameters. And in fact, our approach is less than perfect in terms of precision and recall, but that seems to be the state of the art: "*CloneDetective represents the state-of-the-art [...but the most] important limitations are its inaccuracy and low degree of completeness in detection*" (see [23, p. 276]).

8.3 Outlook

Future work will focus on four areas. First, the proposed definition of model clones will have to be discussed and refined. Second, $\mathsf{MQ}_{\mathsf{lone}}$ can be tuned by a large number of parameter settings that have not yet been fully explored. Third, we want to explore more algorithms, e. g., extending the compared fragments, specializing the algorithm and heuristics for different clone and/or model element types, stacking different algorithms, or using semantic name matching. Also, the experiments so far showed several ways of improving the existing approach, e. g., by removing unavoidable clones. Furthermore, preliminary results not reported in this article indicate that weighing similarity results by the size of the compared model fragments may substantially improve the accuracy of clone recognition.

The most challenging task, however, will be to collect a body of representative and comprehensive UML models with clones, and study clone structures occurring in practices, as these structures determine the quality of any model clone detection approach. At this point, we have eleven domain models from four case studies created by students, with a predictable growth of eight to twelve models and one to three case studies every winter term. Also, we are acquiring models from other colleagues. We would like to take this opportunity to invite the reader to forward any domain models they might have to the author for further study.

References

1. 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.
2. Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., and Selic, B. An MDA Manifesto. *MDA Journal 5* (May 2004), 2–9. available from `bptrends.com/publicationfiles/05-04COLIBMManifesto-Frankel-3.pdf`.
3. Cordy, J. R., Inoue, K., Koschke, R., and Jarzabek, S., Eds. *Proc. 4th Intl. Ws. Software Clones (IWSC)* (2010), ACM. also appeared as 29(2) of ACM SIGSOFT SE Notes.
4. Costagliola, G., and Ko, A., Eds. *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)* (2011), IEEE Computer Society. to appear.
5. Deissenboeck, F., Hummel, B., Juergens, E., Pfaehler, M., and Schaetz, B. Model Clone Detection in Practice. In Cordy et al. [3], pp. 57–64. also appeared as 29(2) of ACM SIGSOFT SE Notes.
6. Deissenboeck, 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. Fish, A., and Störrle, H. Visual qualities of the Unified Modeling Language: Deficiencies and Improvements. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'07)* (2007), P. Cox and J. Hosking, Eds., IEEE Computer Society, pp. 41–49.
8. Proc. IEEE 31st Intl. Conf. Software Engineering (ICSE). In *Proc. IEEE 31st Intl. Conf. Software Engineering (ICSE)* (2009), IEEE Computer Society.
9. Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. Do code clones matter? In ICSE'09 [8], pp. 485–495.

10. Junginger, S., Kühn, H., Strobl, R., and Karagiannis, D. Ein Geschäftsprozessmanagement- Werkzeug der nächsten Generation - ADONIS: Konzeption und Anwendungen. *Wirtschaftsinformatik 42*, 5 (2000), 392–401.
11. 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. Final report on seminar 06301 "Duplication, Redundancy, and Similarity in Software", available at `http://drops.dagstuhl.de/opus/volltexte/2007/970`.
12. Kelter, U., Wehren, J., and Niere, J. A Generic Difference Algorithm for UML Models. In *Proc. Natl. Germ. Conf. Software-Engineering 2005 (SE'05)* (2005), K. Pohl, Ed., no. P-64 in Lecture Notes in Informatics, Gesellschaft für Informatik e.V. pp. 105–116.
13. Kolovos, D. S., Paige, R. F., and Polack, F. A. C. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. Nguyen, H., Nguyen, T., Pham, N., Al-Kofahi, J., and Nguyen, T. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proc. 12th Intl. Conf. Fundamental Approaches to Software Engineering (FASE)* (2009), Springer, pp. 440–455.
21. OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.2 (formal/2009-02-02). Tech. rep., Object Management Group, Feb. 2009.
22. MDA Guide Version 1.0.1. Tech. rep., Object Management Group, June 2003. available at `www.omg.org/mda`, document number omg/2003-06-01.
23. 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 [8], pp. 276–286.

24. Rahm, E., and Bernstein, P. A. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal 10* (2001), 334–350.
25. 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.
26. Roy, C. K., and Cordy, J. R. A Survey on Software Clone Detection. Tech. Rep. TR 541, Queen's University, School of Computing, 2007.
27. Schrepfer, M., Wolf, J., Mendling, J., and Reijers, H. A. The impact of secondary notation on process model understanding. In *The Practice of Enterprise Modeling (PoEM)* (2009), A. Persson and J. Stirna, Eds., Springer Verlag, pp. 161–175.
28. 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.
29. Selic, B. The pragmatics of Model-Driven Development. *IEEE Software 20*, 5 (Sept./Oct. 2003), 19–25.
30. Störrle, H. A PROLOG-based Approach to Representing and Querying UML Models. In *Intl. Ws. Visual Languages and Logic (VLL'07)* (2007), P. Cox, A. Fish, and J. Howse, Eds., vol. 274 of *CEUR-WS*, CEUR, pp. 71–84.
31. Störrle, H. Large Scale Modeling Efforts: A Survey on Challenges and Best Practices. In *Proc. IASTED Intl. Conf. Software Engineering (IASTED-SE'07)* (2007), W. Hasselbring, Ed., Acta Press, pp. 382–389.
32. Störrle, H. A logical model query interface. In *Intl. Ws. Visual Languages and Logic (VLL'09)* (2009), P. Cox, A. Fish, and J. Howse, Eds., vol. 510, CEUR, pp. 18–36.
33. Störrle, H. VMQL: A Generic Visual Model Query Language. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'09)* (2009), M. Erwig, R. DeLine, and M. Minas, Eds., IEEE Computer Society, pp. 199–206.
34. Störrle, H. Structuring very large domain models: experiences from industrial MDSD projects. In Wasowski et al. [40], pp. 49–54.
35. Störrle, H. Towards clone detection in UML domain models. In Wasowski et al. [40], pp. 285–293.
36. Störrle, H. Expressing Model Constraints Visually with VMQL. In Costagliola and Ko [4]. to appear.
37. Störrle, H. On the Impact of Layout Quality to Unterstanding UML Diagrams. In Costagliola and Ko [4]. to appear.
38. Störrle, H. VMQL: A Visual Language for Ad-Hoc Model Querying. *J. Visual Languages and Computing 22*, 1 (Feb. 2011).
39. 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.
40. Wasowski, A., Truscan, D., and Kuzniarz, L., Eds. *Proc. 8th Nordic Ws. Model Driven Engineering (NW-MODE10). Appeared as Proc. 4th Eur. Conf. Sw. Architecture (ECSA'10): Companion Volume, I. Gorton, C.E. Cuesta, M.A. Babar (eds.)* (2010), ACM.
41. Wielemaker, J. SWI Prolog 5.6.46 Reference Manual. Tech. rep., University of Amsterdam, Dept. of Social Science Informatics, 2007. available at `www.swi-prolog.org`.