Differencing UML Models: A Domain-Specific vs. a Domain-Agnostic Method

Rimon Mikhaiel[1], Nikolaos Tsantalis[2], Natalia Negara[1], Eleni Stroulia[1], and Zhenchang Xing[3]

[1] Computing Science Department, University of Alberta, Edmonton, AB, T6G 2E8, Canada
[2] Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, H3G 1M8, Canada
[3] Department of Computer Science, School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417
{rimon,negara,stroulia}@ualberta.ca, tsantalis@cse.concordia.ca, xingzc@comp.nus.edu.sg

Abstract. Comparing software artifacts to identify their similarities and differences is a task ubiquitous in software engineering. Logical-design comparison is particularly interesting, since it can serve multiple purposes. When comparing the as-intended vs. the as-implemented designs, one can evaluate implementation-to-design conformance. When comparing newer code versions against earlier ones, one may better understand the development process of the system, recognize the refactorings it has gone through and the qualities motivating them, and infer high-order patterns in its history. Given its importance, design differencing has been the subject of much research and a variety of algorithms have been developed to compare different types of software artifacts, in support of a variety of different software-engineering activities. Our team has developed two different algorithms for differencing logical-design models of object-oriented software. Both algorithms adopt a similar conceptual model of UML logical designs (as containment trees); however, one of them is heuristic whereas the other relies on a generic tree-differencing algorithm. In this paper, we describe the two approaches and we compare them on multiple versions of an open-source software system.

Keywords: UML, software differencing, software evolution.

1 Introduction

Differencing of software artifacts is a task essential to a variety of software-engineering activities, and a multitude of its instances can be found in a range of well-recognized areas of software-engineering research. Alternative designs are compared to each other in order to recognize their differences and assess their relative merits. Design models are compared against code in order to evaluate implementation-to-design conformance. Newer code versions are compared against earlier ones when submitted to a shared repository, in order to recognize potentially conflicting edits and properly merge them. Code fragments are compared against each other to recognize

R. Lämmel, J. Saraiva, and J. Visser (Eds.): GTTSE 2011, LNCS 7680, pp. 159–196, 2013.

"clones", i.e., lexically and syntactically similar code that could potentially be abstracted into a single "named" and reusable code structure. Component interfaces are matched against queries in order to enable the discovery and selection of reusable components.

In this paper, we explore the problem of object-oriented design differencing. Recognizing the differences between two object-oriented designs is essential for the following tasks.

(a) To understand (at a high level of abstraction) the evolution between two versions of a software system, one may reverse engineer the corresponding design versions and compare the as-implemented design of the software.
(b) To analyze the long-term evolution of a system and its constituent components and recognize interesting restructuring and expansion phases, one can repeatedly perform the above analysis over a sequence of subsequent software versions.
(c) To recognize the progress of the development team towards implementing the software design, one may again reverse engineer the design of the code base and compare it against the intended design of the system.
(d) Finally, to merge out-of-sync versions of software, one has to compare the merge candidates.

We have chosen to focus on design-level differencing because of several reasons. First, design provides a high level, yet information rich, abstraction of the software implementation, essential for understanding complex systems. Second, there is a standard representation of object-oriented design (namely UML and XMI), which is available in the context of many development environments, thus enabling the study of our methods and tools in a broad range of contexts. Third, high-level abstraction makes possible the comparison of design documents (high-level description) against source code (low-level implementation). Finally, adopting logical UML models as the underlying representation of the artifacts to be compared, we have the option of expanding our study to other types of UML models representing requirements (use cases), dynamic behaviors (sequence diagrams) and physical architecture (component diagrams).

Having committed to a particular representation of software, the question becomes how to design an algorithm for comparing instances of this representation. In principle, there are two different methodological approaches to addressing this question. On one hand, one can design an algorithm specific to the adopted representation, aware of the semantics of the modeled elements. The advantages of such domain-specific approaches are that they usually produce intuitive results, since the understanding of the representation semantics is "embedded" in the algorithm design, and their process is usually straightforward to follow and explain. Their major disadvantage is that they are not easy to generalize or to migrate to other representations of software. The alternative is to map the software representation to a more abstract representation (such as strings, trees, or graphs) for which differencing algorithms already exist and to configure these more general algorithms to somehow take into account the semantics of the domain. This approach is clearly more generalizable than domain-specific methods, since one can imagine multiple mappings of the same algorithm to multiple software representations; however it is likely to suffer from unintuitive results since the

complex semantics of the domain have to be abstracted into a set of few elements and their relations.

Our team has been exploring these two alternative methodologies in the context of the PhD theses of Zhenchang Xing [28] and Rimon Mikhaiel [10]. In this paper, we describe in detail *VTracker,* and summarize our understanding of the relative advantages and disadvantages of two algorithms through an extensive comparison on multiple version of an open-source system. The paper is organized as follows. We first present *UMLDiff*, a domain-specific algorithm for differencing UML class models. Next, we discuss *VTracker*, an extended tree-differencing algorithm that can be systematically configured with a domain-specific cost function in order to be applied to tree-like representations in different domains. In presenting the two algorithms, we comparatively discuss their workflow and assumptions with respect to the cost functions they use to compare the various software elements. Next, we present an extensive experiment where both algorithms have been applied to recognize the changes that occurred in multiple successive versions of an open-source system in order to compare their accuracy, efficiency and scalability. Finally, we review a set of use cases where this type of differencing can be applied for a variety of maintenance activities.

2 UMLDiff

UMLDiff is an algorithm designed to compare software systems, in terms of their UML logical models. The algorithm takes as input two directed graphs, $G(V, E)$, corresponding to the models of the systems to be compared. The vertex set, V, of each such graph contains the *elements* of the system's UML logical model; the edge set, E, contains the *relations* among them. The model elements and the possible relations among them are shown in Tables 1 and 2.

Given two versions of a software system and the graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, corresponding to their UML logical-design models, *UMLDiff* essentially maps the two model graphs by computing the intersection and margin sets between (V_1, V_2) and (E_1, E_2). More specifically, $(V_1–V_2)$ and $(E_1–E_2)$ are the sets of removed model elements and relations, $(V_1 \cap V_2)$ and $(E_1 \cap E_2)$ are the sets of the mapped elements and relations, and $(V_2–V_1)$ and $(E_2–E_1)$ are the sets of the added model elements and relations.

UMLDiff is a heuristic tree-differencing algorithm, relying on the fact that the composition relations (see Table 3) induce a spanning tree on the directed graph of the system's UML logical model. The UML semantics guarantees that all model elements can be visited by traversing the containment hierarchy, starting from the top-level subsystem (corresponding to the system as a whole), and that the children of their containing parent are unique in terms of their names. There are four logical levels in which all types of model elements belong (see Table 3): subsystem (including the top-level subsystem) > package > (class, interface) > (attribute, operation). Note that the model elements of type subsystem, package, class and interface may contain same-type elements.

Table 1. Types of Elements in the UML Logical Model

Metaclass <<Stereotype>>	**Description**
Subsystem	A subsystem is a grouping of model elements.
Package	A package is a grouping of model elements (Java specific).
Class	A class declares a collection of attributes, operations and methods, to describe the structure and behavior of a set of objects; it acts as the namespace for various elements defined within its scope, i.e. classes and interfaces.
Interface	An interface is a named set of operations that characterize the behavior of an element.
DataType	A data type is a type whose values have no identity.
Attribute	An attribute is a named piece of the declared state of a classifier, which refers to a static feature of a model element. An attribute may have an initValue specifying the value of the attribute upon initialization.
Operation <<create>> <<initialize>>	An operation is a service that can be requested from an object to effect behavior, which refers to a dynamic feature of a model element.
Method	A method is the implementation of an operation.
Parameter	A parameter is a declaration of an input/output argument of an operation.
Exception	An exception is a signal raised by an operation.
Reception	A reception is a behavioral feature; the classifier containing the feature reacts to the signal designated by the reception feature.

Table 2. Types of Relations among the Elements of a UML Logical Model

Metaclass <<Stereotype>>	**Description**
Generalization	A generalization is a taxonomic relation between a more general element (parent) and a more specific element (child).
Abstraction <<realize>>	An abstraction is a dependency relation; it relates two (sets of) elements representing the same concept.
Usage <<call>><<send>> <<instantiate>> <<read>><<write>>	A usage is a dependency relation in which one element requires another element (or set of elements) for its full implementation or operation.
Association	An association is a declaration of a semantic relation between classifiers that can be of three different kinds: 1) ordinary association, 2) composite aggregate, and 3) shareable aggregate.

Table 3. Composition Relations over the Elements of the UML Logical Models

Element type	**Types of the element's children**
Top-level Subsystem	Subsystem and Package ProgrammingLanguageDataType Class and Interface whose isFromModel=false
Subsystem	Subsystem and Package
Package	Package, Class and Interface
Class	Class and Interface Attribute, Operation, $\text{Operation}_{<<create>>}$, $\text{Operation}_{<<initialize>>}$
Interface	Class and Interface, Operation
Attribute	N/A
Operation	Parameter

2.1 The *UMLDiff* Algorithm

Given two input graphs, *UMLDiff* starts by comparing their vertices, i.e., mapping the elements of the first model to "same" elements of the second model. Once this process has been completed, it proceeds to analyze the relations of the two graphs.

2.1.1 Mapping Elements

UMLDiff traverses the containment-spanning trees of the two compared models, descending from one logical level to the next, in both trees at the same time. It starts at the top-level subsystems that correspond to the two system models and progresses down to subsystems, packages, classes and interfaces, and finally, attributes and operations. At each level, it compares all elements at that level from version 1, e_{i-1}, to all elements of version 2, e_{j-2}, and recognizes pairs of "same" elements, i.e., elements that correspond to the same design-model concept.

Similarity for *UMLDiff* is established on the basis of two criteria: (a) *lexical similarity,* i.e., a metric of the lexical distance between the identifiers of two same-level elements, and (b) *structure similarity,* i.e., a metric of the degree to which the two compared elements are related in the same ways to other elements that have already been established to be the same.

Name similarity is a "safe" indicator that e_1 and e_2 are the same entity: in our experience with several case studies, very rarely is a model element removed and a new element is added to the system with the same name but different element type and different behavior. *UMLDiff* recognizes same-name model elements of the same type first and uses them as initial "landmarks" to subsequently recognize renamed and moved elements.

Within each level, after all same-name elements have been recognized, *UMLDiff* attempts to recognize renamed and/or moved elements at that level. When a model element is renamed or moved – frequent changes in the context of object-oriented refactorings – its relations to other elements tend to remain the same, for the most part. For example when an operation moves, it still reads/writes the same attributes and it calls (and is called by) the same operations. Therefore, by comparing the relations of two same-type model elements, *UMLDiff* infers renamings and moves: the two compared elements are the same, if they share "enough" relations to elements that have already been established to be the same, even though their names (in the case of renamings) and/or their parent (containing) model elements are different (in the case of moves).

The knowledge that two model elements are essentially the same, in spite of having been renamed or moved, is added to the current set of mapped elements, and is used later on to further match other not-yet-mapped elements. This process continues until the leaf level of the two spanning trees has been reached and all possible corresponding pairs of model elements have been identified.

Given two renaming or move candidates, *UMLDiff* computes their structural similarity as the cardinality of the intersection of their corresponding related-element sets (see Section 2.2.2 for details). Given the sets of elements that are connected to the two compared candidates with a given relation type, *UMLDiff* identifies the common

subset of elements that have already been mapped. Therefore, if most of the model elements related to two candidates were also renamed and/or moved and cannot be established as "same", the *UMLDiff* structure-similarity heuristic will fail. If, on the other hand, a set of related elements were renamed or moved but enough model elements related to the affected set remained the "same", it would be possible to recognize this systematic change.

The structure-similarity metric fails when global renamings are applied, i.e., renamings to meet a new naming convention, for example. In such cases, there may be so many elements affected that the initial round of recognizing "same" elements based on name similarity may not produce enough mapped elements, to be used as landmarks for structure similarity. To address this problem, *UMLDiff* can be configured with a user-provided string transformation – introducing a prefix or appending a suffix, or replacing a certain substring – to be applied to the names of the model elements of one of the compared versions, before the differencing process. To further accelerate the recognition of "same" elements, *UMLDiff* propagates operation renamings along the inheritance hierarchy, i.e., it assumes that if an operation o_1 in a class c_1 has been renamed to o_2, then all its implementations in the subclasses of c_1 have also been similarly renamed.

Finally, as each round of recognition of "same" elements based on structure similarity establishes more landmarks on the basis of which new elements can be recognized as structurally similar, *UMLDiff* can be configured to go through multiple rounds of renaming and move identification, until no more new renamed and/or moved elements can be found or it finishes the user-specified number of iterations.

2.1.2 Mapping Relations

Once *UMLDiff* has completed mapping the sets of model elements, V_1 and V_2, it proceeds to map the relation sets, E_1 and E_2, by comparing the relations of all pairs of model elements (v_1, v_2), where v_2=null if v_1 is removed and v_1=null if v_2 is added. The relations from (to) a removed model element are all removed and the relations from (to) an added model element are all added. For a pair of mapped elements (v_1, v_2), they may have matched, newly added, and/or removed relations. Note that a removed (added) relation between two model elements does not indicate any of the elements it relates being removed (added).

Finally, *UMLDiff* detects the redistribution of the semantic behavior among operations, in terms of usage-dependency changes, and computes the changes to the attributes of all pairs of mapped model elements.

2.1.3 Configuration Parameters

The *UMLDiff* differencing process is configured through the following set of parameters.

1. The *LexicalSimilarityMetric* specifies which of three alternative lexical-similarity metrics (Char-LCS, Char-Pair, and Word-LCS) will be used by *UMLDiff*.

2. The *RenameThreshold* and *MoveThreshold* specify the minimum similarity values between two model elements in the two compared versions in order for them to be considered as the same conceptual element renamed or moved. *UMLDiff* allows multiple rounds (*MaxRenameRound* and *MaxMoveRound*) of renaming and/or move identification in order to recover as many renamed and moved entities as possible.
3. The *ConsiderCommentSimilarity* parameter defines whether the similarity of the comments of the model elements should also be taken into account when comparing two elements, if the compared elements have an initial overall similarity value above the *MinThreshold*. This threshold prevents model elements with very low name- and structure-similarity from qualifying as renamings or moves just because of their similar comments.
4. The *ConsiderTransclosureUsageSimilarity* parameter controls whether the similarity of the transitive usage dependencies between two compared operations may also be used to assess their structural similarity.
5. At the end of the differencing process, *UMLDiff* can be instructed whether or not to compute the usage dependency changes for all model elements and analyze the redistribution of operation behavior.

2.2 Assessing Similarity

In the above section, we have described how *UMLDiff* maps elements relying on two heuristics – lexical and structure similarity. In this section we delve deeper on the details of how exactly lexical and structure similarity are computed. The equations specifying these computations are intuitively motivated and have been tuned through substantial experimentation. These computations are fundamentally heuristic, tailored to the idiosyncrasies of the UML domain and our intuitions and understanding of the practices of developers in naming identifiers.

2.2.1 Lexical Similarity

To assess the similarity of the identifiers of (and the textual comments associated with) two compared model elements, *UMLDiff* integrates three metrics of string similarity: (a) the longest common character subsequence (Char-LCS); (b) the longest common token subsequence (Word-LCS); and (c) the common adjacent character pairs (Char-Pair). All these metrics are computationally inexpensive to calculate, given the usually small length of the names and comments of model elements. They are also case insensitive, since it is common to misspell words with the wrong case or to modify them with just case changes. They are all applicable to name similarity, while only Char-LCS and Word-LCS may be applied to compute comment similarity. Irrespective of the specific metric used, let us first describe what exactly *UMLDiff* considers as the "identifier" of each model-element type.

The lexical similarity of operations is calculated as the product of their identifier similarity and their parameter-list similarity. In turn, the similarity of two parameter

lists is computed based on the Jaccard coefficient of the two bags of data types of the operations' parameters, i.e. the intersection of two bags of parameter types divided by the union of two bags of parameter types.

For packages, we split package names into a set of words by ".", and then compute the lexical similarity of packages using the similarity equations defined below. The similarity of the comments associated with two model elements is only consulted when both elements have associated comments (i.e., the *UMLDiff* parameter *ConsiderCommentSimilarity* is true) and the initial overall similarity metric between these elements is greater than the *UMLDiff* parameter *MinThreshold*.

The longest common character subsequence (Char-LCS) algorithm [15] is frequently used to compare strings. Word-LCS applies the same LCS algorithm, using words instead of characters as the basic constituents of the compared strings. The names of model elements are split into a sequence of words, using dots, dashes, underscores and case switching as delimiters. Comments are split into words using space as the sole delimiter. The actual metric used for assessing LCS-similarity is shown in Equation 1.

$$\text{Char/Word-LCS}(s_1, s_2) = 2 * \text{length}(\text{LCS}(s_1, s_2)) / (\text{length}(s_1)+\text{length}(s_2)),$$

where LCS() and length() is based on the type of token considered, i.e., characters or words.

Equation 1

LCS reflects the lexical similarity between two strings, but it is not very robust to changes of word order, which is common with renamings. To address this problem, we have defined the third lexical-similarity metric in terms of how many common adjacent character pairs are contained in the two compared strings. The *pairs(x)* function returns the pairs of adjacent characters in a string *x*. By considering adjacent characters, the character ordering information is, to some extent, taken into account. The Char-Pair similarity metric, which is a value between 0 and 1, is computed according to Equation 2.

$$\text{Char-Pair}(s_1, s_2) = 2 * |\text{pairs}(s_1) \cap \text{pairs}(s_2)| / (|\text{pairs}(s_1)|+|\text{pairs}(s_2)|).$$

Equation 2

2.2.2 Structure Similarity

Table 4 lists the relations that *UMLDiff* examines to compute the structure similarity between two model elements of the same type. The top-level subsystems, corresponding to the two compared versions of a UML logical model, are always assumed to match. The structure similarity of subsystems, packages, classes and interfaces is determined based on (a) the elements they contain, (b) the elements they use, and (c) the elements that use them. The structure similarity of attributes is determined by the operations that read and write them, and their initialization expressions. The structure similarity of operations is determined by the parameters they declare, their outgoing usage dependencies (including the attributes they read and write, the operations they call, and the classes/interfaces they create), and their incoming usage dependencies (including the attributes (through their *initValue*) and the operations that call them).

Table 4. The UML relations for computing structure similarity

Element type	Type of relations
Subsystem	[namespace – ownedElement] Incoming and outgoing usage
Package	[namespace – ownedElement] Incoming and outgoing usage
Class, Interface	[namespace – ownedElement] and [owner – feature] Incoming and outgoing usage
Attribute	$Usage_{<<read>>}$, $Usage_{<<write>>}$ and inherent Attribute.initValue
Operation	[BehaviorFeature – parameter] and [typedParameter – type] Outgoing usage: $Usage_{<<read>>}$, $Usage_{<<write>>}$, $Usage_{<<call>>}$, $Usage_{<<instantiate>>}$ Incoming usage: $Usage_{<<call>>}$

The structure similarity of two compared elements is a measure of the overlap between the sets of elements to which the compared elements are related. The intersection of the two related-element sets contains the pairs of model elements that are related to the compared elements (with the same relation type) and have already been mapped. In effect, this intersection set incorporates knowledge of any "known landmarks" to which both compared model elements are related.

Given two model elements of the same type, v_1 and v_2, let Set_1 and Set_2 be their related-element sets, the structure similarity between v_1 and v_2 according to a given group of relations is a normalized value (between 0 and 1) as computed according to Equation 3.

StructureSimilarity = matchcount / (matchcount + addcount + removecount), where the matchcount, addcount, and removecount are the cardinalities of $[Set_1 \cap Set_2]$, $[Set_2 - Set_1]$, $[Set_1 - Set_2]$ respectively.

Equation 3

For a usage dependency, its *count* tag, which indicates the number of times that it appears between the client and supplier elements, is used to compute its matchcount, addcount, and removecount.

The similarity of the parameter lists of two operations is based on the names and types of their parameters. The computation of parameter-list similarity is insensitive to the order of parameters. For non-return parameters, if none of the two operations is overloading, the matchcount for a pair of same-name parameters is 1. If any of the two compared operations is overloading, the types of the two same-name parameters is further examined, in order to distinguish the overloading methods from each other, which often declare the same-name parameters but with different parameter types. In the case of overloading, if the same-name parameters are of mapped types, their matchcount is 1; otherwise, their matchcount is 0.5. For the return parameters, if their types are mapped, the matchcount is 1; else it is set at 0. If the type of the return parameter of both operations is void, the matchcount for the return parameter is 0.

The similarity of the *initValue* of two compared attributes is computed in the same way as the outgoing usage similarity between two operations. The initValue-similarity value is added to the overall matchcount of the $Usage_{<<write>>}$ similarity between two attributes.

Determining the similarity when both related model-element sets are empty is challenging, when, for example, two operations are not called by any other operations. In such cases, setting the structure similarity to be by default 0 or 1 is not desirable: without any explicit evidence of similarity, assuming that the structure is completely the same or completely different may skew the subsequent result. Therefore, in such cases, *UMLDiff* uses the name similarity with an increasing exponent. The effect is dampened as more empty sets are encountered. For example, when computing the structure similarity of two operations in the order of their parameter-list, outgoing usage and incoming usage similarities, if the two compared operations declare no parameters, have return type void, and have no outgoing and incoming usage dependencies, *UMLDiff* returns *name-similarity*^{1} for comparing parameter-list similarity, *name-similarity*^{2} for outgoing usage similarity, and *name-similarity*^{3} for incoming usage similarity.

2.2.3 Overall Similarity Assessment

Given two model elements e_1 and e_2 of the same type, their overall similarity metric, used for determining potentially renamed and moved model elements, is computed according to the Equation 4, below.

SimilarityMetric=(lexical-similarity+$\Sigma_N$structure-similarity)/(lexical-similarity+N), where lexical-similarity = name-similarity + comment-similarity, and *N* is the number of different types of structure similarities computed for a given type of model elements, as defined in Table 2.

Equation 4

The value of $\Sigma_N$structure-similarity is adjusted in the following cases.

When comparing two operations, if any of them is overloaded, $\Sigma_N$structure-similarity is multiplied by the parameter-list similarity of the compared operations in order to distinguish the overloading operations from each other, which often have similar usage dependencies but with different parameters.

When determining the potential moves of attributes and operations, if the declaring classes/interfaces of the compared attributes/operations are not related through inheritance, containment, or usage relations, the value of $\Sigma_N$structure-similarity is multiplied by the overall similarity of the classes in which the compared attributes/operations are declared, and divided by the product of the numbers of all the not-yet-mapped model elements with the same name and type as the two compared elements. This is designed to improve the low precision when identifying attribute and operation moves.

UMLDiff uses two user-defined thresholds (*RenameThreshold* and *MoveThreshold*): two model elements are considered as the "same" element renamed or moved when their overall similarity metric is above the corresponding threshold. If, for a given element in one version, there are several potential mappings above the user-specified threshold in the other version, the one with the highest similarity score is chosen. The higher the threshold is, the stricter the similarity requirement is. The smaller the threshold is, the riskier the renamings-and-moves recognition process is.

3 VTracker

The *VTracker* algorithm is designed to compare XML documents, based on a tree-differencing paradigm. It calculates the minimum edit distance between two labeled ordered trees, given a cost function for different edit operations (e.g. change, deletion, and insertion). Essentially, *VTracker* views XML documents as partially ordered trees, since XML elements contain other XML elements and the order of contained elements within a container does not matter, unless these elements are contained in a special ordered container. Given that UML logical models can be represented in XMI, i.e., an XML-based syntax, the problem of UML logical-model differencing can be reduced to XML-document differencing and *VTracker* can be applied to it.

VTracker is based on the Zhang-Shasha's tree-edit distance [30] algorithm, which calculates the minimum edit distance between two trees T_1 and T_2[1], given a cost function for different edit operations (e.g. change, deletion, and insertion) in complexity of $O(|T_1|^{3/2}|T_2|^{3/2})$, according to the analysis of Dulucq and Tichit [1].

Intuitively, given two trees, the Zhang-Shasha algorithm identifies the minimum cost of mapping the nodes of the two trees to each other, considering the following three options, illustrated in Figure 1.a:

(a) the cost of mapping the root nodes of the two trees plus the cost of mapping the remaining forests to each other (assuming that the root nodes of the two trees are comparable);
(b) the cost of deleting the root of the first tree plus the cost of mapping the remaining forest to the entire second tree (assuming that the root of the first tree was newly inserted in the second tree); and
(c) the cost of deleting the root of the second tree plus the cost of mapping the entire first tree against the remaining forest of the second tree (assuming that the root of the first tree is missing in the second tree).

The *VTracker* algorithm, for calculating the edit distance between two trees rooted by nodes *x* and node *y* respectively, is shown in pseudocode in Algorithm 1. The algorithm assumes that nodes are numbered in a post-order manner where a parent node is visited after all its children, from left to right, have been recursively visited. The process, as shown in lines 5-8, starts by determining the span of each node (x and y); the span of node x includes all the nodes starting at the left-most child of x to x, the root, plus a "dummy" node, which represents the *void* node given index zero while the left-most child at index one. The algorithm proceeds to progressively calculate the edit distance between portions (forests) from both trees. For example, fdist[i][j] is the distance between the first forest (including all the nodes in the first tree up to and including node with index i) and the second forest (including all the nodes in the second tree up to and including node with index j). Then, the process keeps adding a single node on each of the compared forests (lines 10 to 13) and assessing the cost, until it reaches the last point where both sides are not forests anymore but the complete trees.

[1] We use T_1 and T_2 to refer to the trees and the number of their nodes, at the same time.

```
Input: T1 and T2 trees
01 DECLARE matrix tdist with size [|T1|+1] * [|T2|+1]
02 DECLARE matrix fdist with size [|T1|+1] * [|T2|+1]

03 FUNCTION treeDistance (x , y)
04 START
05  lmx = lm1(x)                                  // left most node of x
06  lmy = lm2(y)                                  // left most node of y
07  span1 = x - lmx + 2                         //size of sub-tree x + 1
08  span2 = y - lmy + 2                        //size of sub-tree y + 1
09  fdist[0][0] = 0
10  FOR i = 1 TO span1 - 1                        // set the first column
11   fdist[i][0] = fdist[i-1][0] + cost(k,-1,i,j)
12  FOR j = 1 TO span2 - 1                         // set the first row
13   fdist[0][j] = fdist[0][j-1] + cost(-1,l,i,j)

14  k = lmx
15  l = lmy
16  FOR i = 1 TO span1 - 1
17   FOR j = 1 TO span2 - 1
18    IF lm1(k) = lmx and lm2(l) = lmy
19    THEN                                        // tree edit distance
20     fdist[i][j] = min(fdist[i-1][j] + cost(k,-1,i,j),
                         fdist[i][j-1] + cost(-1,l,i,j),
                         fdist[i-1][j-1] + cost(k,l,i,j))
21     tdist[k][l] = fdist[i][j]
22    ELSE                                        // forest edit distance
23     m = lm1(k) - lmx
24     n = lm2(y) - lmy
25     fdist[j][j] = min(fdist[i-1][j] + cost(k,-1,i,j),
                         fdist[i][j-1] + cost(-1,l,i,j),
                         fdist[m][n] + tdist(k,l,i,j))
26    l++
27   k++
28  RETURN tdist[x][y]
29 END
```

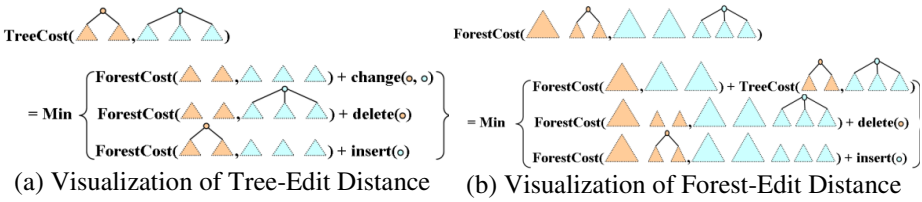
Algorithm 1: The Zhang-Sasha Tree Comparison

At line 9, the algorithm starts by initializing fdist[0][0], i.e., the cost of transforming a void forest into another void forest, to zero. In lines 10 and 11 it calculates the deletion costs of various forests of the first tree, which it progressively leads to calculating the cost of deleting the whole first tree. Similarly, the algorithm calculates the insertion costs in lines 12 and 13. At this point it has calculated the cost of mapping the two trees through the drastic change of deleting all the nodes of the first one and adding all the nodes of the second.

Then, beginning at line 18, the algorithm starts adding one node to each tree and calculating the distance between the resulting forests. In each step, if both sides have one full sub-tree, it applies the tree distance mechanism; otherwise it uses the forest edit distance mechanism (illustrated in Figure 1.b), where it chooses the minimum cost option of the three below:

- The cost of mapping node x to node y plus the cost of matching the remaining forests to each other.
- The cost of deleting node x plus the cost of matching remaining forest of first tree against the entire second tree.

- The cost of inserting node y plus the cost of matching entire first tree against remaining forest of the second tree.



(a) Visualization of Tree-Edit Distance (b) Visualization of Forest-Edit Distance

Fig. 1. Visualization of Zhang-Shasha algorithm [30]

VTracker extends the Zhang-Shasha algorithm in four important ways. First, it uses an affine-cost policy, which adjusts the cost of each operation if it happens in the vicinity of many similar operations. The affine-cost computation algorithm is discussed in Section 3.1.

Second, unlike the Zhang-Shasha algorithm, which assumes "pure" tree structures, *VTracker* allows for cross-references between nodes of the compared trees, which is essential for comparing XML documents that use the ID and IDREF attributes. *VTracker* considers the existence of these references in two different situations during the matching process. First, it considers referenced elements as being a part of the referring elements' structure (see Section 3.2); when two nodes are being compared, *VTracker* considers all their children irrespective of whether they are defined in the context of their parent nodes or referenced by them. Additionally, through its "context-aware matching" process, *VTracker* considers not only the internal structure of the compared elements but also the context in which they are used, namely the elements by which they are being referenced.

Third, in a post-processing step, *VTracker* applies a simplicity-based filter to discard the more unlikely solutions from the solution set produced during the tree-alignment phase (see Section 3.3).

Finally, in addition to being applied with the default cost function that assigns the same cost to addition/deletion/change operations, *VTracker* can be configured with a domain-specific cost function (see Section 3.4) constructed through an initial boot-strapping step where *VTracker* with the default cost function is applied to comparing the forest of elements from the XML Schema Definition of the domain to itself.

3.1 Cost Computation

The original Zhang-Shasha algorithm assumes that the cost of any deletion/insertion operation is independent of the context in which the operation is applied: the cost of a node insertion/deletion is the same, irrespective of whether or not that node's children

are also deleted or inserted. As a result, the Zhang-Shasha algorithm considers as equally expensive two different scripts with the same number and types of edits, with no preference to the script that may include all the changes within the same locality. Such behavior is unintuitive: a set of changes within the same sub-tree is more likely than the same set of changes dispersed across the whole tree. Since the parent-child relation within the tree is likely to represent a semantic relation in the domain, whether it is composition (the parent contains the child), or inheritance (the parent is a super-type of the child), or association (the parent uses/refers to the child), it is more likely than not that changes in one participant of the relation will affect the other. This is why changes are likely to be clustered together around connected nodes, as opposed to "hitting" a number of unrelated nodes.

In order to produce more intuitive tree-edit sequences, *VTracker* uses an affine-cost policy. In *VTracker*, a node's deletion/insertion cost is context sensitive: if all of a node's children are also candidates for deletion, this node is more likely to be deleted as well, and then the deletion cost of that node should be less than the regular deletion cost. The same is true for the insertion cost.

As shown in the Algorithm 2 below, the cost function accepts four parameters. The first two parameters, *x* and *y*, represent the absolute indexes of the two nodes being considered within the two full trees; the other two parameters, *i* and *j*, representing the local their indexes within the two sub-trees being considered that help to determine the edit operation context. A delete operation is denoted by *y=-1*, and an insert operation is denoted by *x=-1* correspondingly; otherwise, it is matching operation and the objective is to assess how much it will cost to transform a node x to node y. As shown in *GetDeletionCost* function to assess the cost of deleting a certain node, the node is checked to be eligible for an affine discounted cost; otherwise the standard edit cost is used. The *GetInsertionCost* function is similar to the deletion one.

```
FUNCTION Cost (x, y, i, j)
START
 IF y = -1
 THEN RETURN GetDeletionCost (x, i, j)
 ELSEIF x = -1 RETURN GetInsertionCost (y, i, j)
 ELSE RETURN MappingCost (x, y, i, j)
 ENDIF
END

FUNCTION GetDeletionCost (x, i, j)
START
 IF IsDeleteAffineEligible(i, j)
 THEN RETURN DISCOUNTED_DELETION_COST  // the whole tree is deleted
 ELSE RETURN STANDARD_DELETION_COST
 ENDIF
END
```

Algorithm 2: Calculating Costs

Algorithm 3 explains the logic of calculating the cost of transforming node *x* to node *y*, i.e., the cost of mapping nodes x and y. Normally, a *NodeDistance* function is used to reflect the domain logic of assessing the cost of node *x* being transformed to node *y*. However, if any of the two nodes *x* or *y* has reference to another node, a different mechanism is used. This mechanism follows the reference to the referred-to

node. Consider for example the case where node *x* has no references, while node *y* is a reference to node *z*. In order to assess the similarity between nodes *x* and *y*, we actually need to assess the similarity between node *x* and node *z*. To that end, the *treeDistance* algorithm, described in Algorithm 1, is used to assess the similarity between the sub-tree rooted at *x* and the sub-tree rooted at *z*. This mechanism is explained in more details in Section 3.2.

```
FUNCTION MappingCost (x,y,i,j)
START
   newX = x
   newY = y
   IF x has a reference
   THEN newX = referenced Id
   ENDIF
   IF y has a reference
   THEN newY = referenced Id
   ENDIF
   IF x <> newX OR y <> newY
   THEN RETURN (treeDistance(newX,newY)/
                (TreeDeletionCost(newX)+TreeInsertionCost(newY)) )*
                STANDARD_CHANGE_COST
   ELSE RETURN NodeDistance(x,y)
   ENDIF
END
```

Algorithm 3: Cost, in the presence of References

```
FUNCTION IsDeleteAffineEligible (i,j)
START
 IF y = 0
 THEN      // the whole tree is to be deleted
  RETURN true
 ELSE      // Cost of matching sub-forest is the actual cost minus
           // Cost of matching the remaining forests to each other
   CostSubForest = fdist [i-1][j] - fdist [lm1(i)-1][j]
           // Cost of deleting everything minus
           // Cost of matching the remaining forests to each other
   CostDelSubForest = fdist [i-1][0] - fdist [lm1 (i)-1][0]

  IF costSubForest = costDelSubForest
   RETURN true
  ELSE
 RETURN false
END
```

Algorithm 4: Affine Costs

3.2 Reference-Aware Edit Distance

Tree-edit distance algorithms only consider node-containment relationships, i.e., parent nodes containing children nodes. *VTracker*, designed for XML documents, is not a pure tree-differencing algorithm; it is aware of other relations between XML elements that are represented as additional references between the corresponding tree nodes. This feature is very important, since most XML documents reuse element definitions thus implying references from an element to the original element definition. The Zhang-Shasha simply ignores such references. In *VTracker* such reference structure is considered in an integrated manner within the tree-edit distance calculation process.

A typical interpretation of such references is that the referenced element structure is meant to be entirely copied under at the reference location; but, to avoid potential inconsistencies through cloning and local changes, elements are reused through a reference to one common definition. *VTracker* compares tree nodes by traversing the containment structure until it encounters a reference. It then recursively follows the reference structure as if it was a part of the current containment structure, until it reaches a previously examined node; then it backtracks recording all the performed calculations, for future use by other nodes referring to the same node.

The question then becomes "how should the cost function be adjusted in order to compute the differences of two nodes in terms of the similarities and differences of the elements they contain and refer to?" As shown in Algorithm 3 above, the definition of the cost function is changed when one of the nodes is a reference to another node. If any or both nodes are references (i.e., have nothing but references), then the cost of changing one into the other is the tree edit-distance between the referenced tree structures. Let's assume that node x refers to node x' and node y refers to node y'. The cost of changing node x to node y is the tree-edit distance between the sub-tree rooted at x' against the sub-tree rooted at y'. Additionally, a normalization step is essential here because the tree-edit distance between x' and y' can vary according to the size of the two trees. Our approach divides the calculated edit distance between the two referenced sub-trees by the cost of deleting both of them which is the maximum possible cost. In this sense, the normalized cost is always ranging from 0 (in case of perfect match) to 1 (in case of totally different structures). Finally, the normalized edit distance is scaled against the maximum possible cost of change, i.e. a normalized cost of 1.0 should be scaled to the maximum cost of changing two nodes to each other. This step is necessary to ensure that the calculated change cost is in harmony with other calculated change costs.

In addition to taking into account efferent relations, i.e., references from the compared nodes to other nodes, *VTracker* also considers the afferent relations of the compared elements, i.e., their "usage context" by nodes that refer to the compared elements. In a post-calculation process, usage-context distance measures are calculated and combined with standard tree-edit distance measures into a new context-aware tree edit distance measure. For each two nodes x and y, we established two sets, $context_1(x) = \{v \mid v \rightarrow x\}$ and $context_2(y) = \{w \mid w \rightarrow y\}$, that include the nodes from which x and y are referenced, respectively. Now, the usage-context distance between x and y is calculated as the Levenshtein edit distance [6] between these elements, where the distance between any two elements is the tree edit distance between these two sub-trees, and the final result is called the *usage context distance* between x and y. Finally, the consolidated context-aware tree edit distance measure is the average between the usage context distance and the tree edit distance measure.

3.3 Simplicity Heuristics

Frequent times, the differencing process may be unable to produce a unique edit script as there may be multiple scripts that transform one tree to the other with the same minimum cost. *VTracker* uses three simplicity heuristics, to discard the more unlikely solutions from the result set.

The path-minimality criterion eliminates "long paths". When there is more than one different path with the same minimum cost, the one with the least number of deletion and/or insertion operations is preferable.

The vertical simplicity heuristic eliminates any edit sequences that contain "non-contiguous similar edit operations". Intuitively, this rule assumes that a contiguous sequence of edit operations of the same type essentially represents a single mutation or refactoring on a segment of neighboring nodes. Thus, when there are multiple different edit-operation scripts with the same minimum cost, and the same number of operations, the one with the least number of changes (refractions) of edit-operation types along a tree branch is preferable.

Finally the horizontal simplicity criterion is implemented by counting the number of *horizontal refraction points*, found when a node suffers an edit operation different from the one applied to its sibling. Therefore, a solution where the same operation is applied to (most of) a node's children is preferable to another where the same children suffer different types of edit operations.

3.4 Schema-Driven Synthesized Cost Function

The *VTracker* algorithm is generic, i.e., it is designed to compare XML documents in general and not XMI documents specifically. However, in order to produce accurate solutions that are intuitive to domain experts, *VTracker* needs to be equipped with a domain-specific cost function that captures the understanding of subject-matter experts of what constitutes similarity and difference among elements in the given domain. Lacking such knowledge, a standard cost function can always be used as a default, which may however sometimes yield less accurate and non-intuitive results. To address the challenge of coming up with a "good" domain-specific cost function, we have developed a method for synthesizing a cost function from the domain's XML schema, relying on the assumption that the XML schema captures in its syntax a (big) part of the domain's semantics. Essentially, *VTracker* assumes that the designers of the domain schema use their understanding of the domain semantics to identify the basic domain elements and to organize related elements into complex ones.

In addition to the domain-specific or default cost functions, *VTracker* uses more cost functions to handle node-level cost assessment. For example, *VTracker* uses a Levenshtein string edit distance [6] to measure the distance between any two literal values like two node names, attribute names or values, text node contents, etc.

4 Comparison of the *UMLDiff* vs. *VTracker* Methodologies

UMLDiff and *VTracker* have both been applied to the task of recognizing design-level differences between subsequent system versions. In this section we review some interesting methodological differences between the two of them.

They both conceptualize logical-design models of object-oriented software as trees. The parent-child relationship between tree nodes corresponds (a) to the instances of the composition relations in *UMLDiff* and (b) to the XMI containment relations, in *VTracker*. The two sets of relations are essentially the same. Practically, *UMLDiff* is applied to a database of "design facts" extracted through a process that analyzes a system's source code; therefore *UMLDiff* always takes into account the exact same relations. *VTracker,* on the other hand, takes as input two XML documents of any type; to be applied to the task of UML model comparison, in principle, it should be provided with the XMI representation of the model. In practice, however, *VTracker's* computation requires too much memory and therefore it cannot be applied to the complete raw XMI representations of large systems. Therefore it has to be applied to a filtered version of XMI and therefore care has to be given on what elements of the XMI syntax are preserved to be considered by *VTracker*. Through experimentation during the development of the WebDiff system [14], we have discovered that *VTracker* works well when applied to XML composition models of single classes, and inheritance models. When multiple classes are compared at the same time, the mapping of tree elements becomes more complex and the computation tends to become impractical. Performance is at the crux of the difference between the two approaches. By restricting itself to a consistent representation of the same design facts, *UMLDiff* can make assumptions about what to consider comparing and how. *VTracker* does not always get applied to the same types of XML documents, and, as a result, in its particular application, one has to trade off "richness" of the model representation against efficiency.

Both *UMLDiff* and *VTracker* can be aware of additional types of relations, like association and inheritance, between logical-model elements. *UMLDiff* exploits these relations while calculating the structure-similarity metric between same-type elements that are considered as candidates for move or renaming. With *VTracker* there are two options. Assuming containment as the primary relation defining the tree structure, additional edges between model elements can be introduced to reflect these other relations. This approach enables *VTracker* to consider these relations through its usage-context and reference-aware matching features; however, it has a substantial negative impact on its performance. In our experimentation with *VTracker* to date, we have developed parallel representations of the logical model, each one considering one of these relations separately, resulting in separate containment, inheritance and association trees, each one to be compared with the corresponding tree of the second logical model.

UMLDiff and *VTracker* exhibit interesting similarities and differences in terms of their similarity/cost functions for comparing model elements.

- They both combine metrics of lexical and structure similarity.
- We have experimented with a variety of lexical similarity metrics for comparing identifiers in *UMLDiff*. *VTracker*, by default, assigns 0 to the distance between two elements when their labels (i.e., identifiers) are the same and 1 when not and can be configured to use the Levenshtein distance [6] for these labels.

- The function for *UMLDiff*'s structural similarity assessment was "hand crafted" after much experimentation. *VTracker*'s cost function is by default very simple (all change operations have the same cost) and has been extended with affine policy and domain-specific weight calculation.

To study in detail the similarities and differences of the two approaches we performed an extensive experiment, where the two methods have been applied to recognize the changes that occurred in multiple successive versions of an open-source system. More specifically, the experiment is driven by three research questions:

1. How does the generic differencing algorithm perform (in terms of precision and recall) compared to the tailor-made one in the examined differencing problem?
2. Is the generic differencing algorithm efficient and scalable in the examined differencing problem?
3. Does the additional effort required for the configuration of the generic differencing algorithm make it an acceptable solution for the examined differencing problem?

To answer the aforementioned research questions we performed a direct comparison of *VTracker* with *UMLDiff* against a manually obtained gold standard. In the following subsections, we describe in detail the process that has been applied in order to conduct this experiment.

4.1 Specification of XML Input for *VTracker*

As we have already mentioned above, *VTracker* is a tree-differencing algorithm, potentially able to handle any kind of XML documents. Nevertheless, the particulars of the XML schema of the documents to be compared can have substantial implications for the accuracy and efficiency of *VTracker*. Therefore, it is very important to come up with an appropriate XML representation of the design elements and relationships in an object-oriented software system. To this end, we have divided the object-oriented design model to three distinct hierarchical structures, implied by the three different dependency relationships (design aspects) specified by the Unified Modeling Language (UML).

- *Containment*: A hierarchical structure representing the containment relationships between a class and its members (i.e., operations and attributes declared within the body of the class).
- *Inheritance*: A hierarchical structure representing inheritance relationships (including both generalization and realization relationships) between classes.
- *Usage*: A hierarchical structure representing the usage dependencies among an operation and other operations and/or attributes (i.e., operation calls and attribute accesses within the body of the operation).

We applied *VTracker* on the three aforementioned design aspects separately for each class of the examined system. This *divide-and-conquer* approach leads to the construction of XML trees with a smaller number of nodes compared to the

alternative approach of using a single XML tree for all design aspects together or for all the classes of the examined system. A direct consequence of this approach is the improvement of efficiency due to the significant reduction in the size of the trees being compared. An indirect consequence is the improvement of accuracy, since the possibility of extracting incorrect node matches is smaller when the number of node combinations that have to be compared is smaller.

The process of generating the XML input files for *VTracker* is performed in the following steps:

1. The source code of the two compared versions is parsed and analyzed in order to extract the structure and the relationships between the source code elements of the underlying design models.
2. For each class being present in both versions, we generate a pair of XML files (one for each compared version) for each one of the examined design aspects (i.e., containment, inheritance and usage).

Figure 2 shows a pair of generated XML files regarding the containment design aspect of class `PaintItem` in versions 1.0.5 and 1.0.6 of JFreeChart. The hierarchical structure of the XML files represents the containment relationships that exist between the source-code elements declared in the given class. For example, operation `PaintItem()` and attributes `paint` and `value` are members of class `PaintItem`, while parameters `paint` and `value` belong to operation `PaintItem()`.

```
−<VirtualRoot name="VirtualRoot">
  −<Class className="org.jfree.chart.renderer.LookupPaintScale.PaintItem" isAbstract="false" isInterface="false">
    −<Operation operationName="PaintItem" param1="java.lang.Number" param2="java.awt.Paint" visibility="public"
       <Parameter paramKind="in" paramName="value" paramType="java.lang.Number"/>
       <Parameter paramKind="in" paramName="paint" paramType="java.awt.Paint"/>
     </Operation>
     <Attribute attrName="paint" type="java.awt.Paint" visibility="package"/>
     <Attribute attrName="value" type="java.lang.Number" visibility="package"/>
   </Class>
 </VirtualRoot>
```

(a) Version 1.0.5

```
−<VirtualRoot name="VirtualRoot">
  −<Class className="org.jfree.chart.renderer.LookupPaintScale.PaintItem" isAbstract="false" isInterface="false">
    −<Operation operationName="PaintItem" param1="double" param2="java.awt.Paint" visibility="public">
       <Parameter paramKind="in" paramName="value" paramType="double"/>
       <Parameter paramKind="in" paramName="paint" paramType="java.awt.Paint"/>
     </Operation>
     <Attribute attrName="paint" type="java.awt.Paint" visibility="package"/>
     <Attribute attrName="value" type="double" visibility="package"/>
   </Class>
 </VirtualRoot>
```

(b) Version 1.0.6

Fig. 2. XML representation of class `PaintItem` for containment

In Figure 2, one can observe that the parameter types of an operation are represented both as attributes of the `Operation` node, as well as attributes of the `Parameter` child nodes. The motivation behind this apparent duplication of information is to further improve the accuracy of *VTracker* when trying to match overloaded operations (i.e., operations having the same name but a different number or types of parameters). By including them as attributes of the `Operation` node, we give to these attributes an increased weight (compared to the weight that they normally have as attributes of the `Parameter` child nodes) and thus we can avoid the problematic situation of mapping incorrectly a set of overloaded operations in the first tree to the corresponding set of overloaded operations in the second tree.

Figure 2 shows that two changes occurred in class `PaintItem` between versions 1.0.5 and 1.0.6. The type of the attribute `value` as well as the type of the parameter `value` in operation `PaintItem()` have been changed from `Number` to `double` (the changes are highlighted in yellow). The XML files regarding the inheritance and usage design aspects are structured in a similar manner.

4.2 Configuration of *VTracker*

The configuration of *VTracker* plays an important role on the accuracy of the technique, since it affects the weights assigned to the attributes of the nodes during the pair-wise matching process. The configuration process is very straightforward, since it only requires the specification of two properties.

The first property is `idAttributeName` for which we have to specify the most important attribute (i.e., *id attribute*) for each type of node in the compared trees. The specified attributes are assigned a higher weight compared to the other attributes of each node type. Practically, this means that if the ID attribute of a node is changed, then the two versions of the node are considered less similar than if another attribute was changed.

The second property is `changePropagationParent` for which we have to specify the node types that should be reported as changed if at least one of their child nodes is added, removed or changed. This feature allows us to identify that a node has changed because of changes propagated from its children, even if the parent node itself is unchanged. For example, an operation node should be considered as changed if one of its parameters has been renamed even if this specific change has no effect on the attributes of the operation node.

Table 5 shows the configuration properties that we have specified for the XML files corresponding to the containment design aspect (as shown in the example of Figure 2).

Table 5. Configuration of *VTracker* for the containment design aspect

Property	Value(s)
idAttributeName	Class => *className* Operation => *operationName* Parameter => *paramName* Attribute => *attrName*
changePropagationParent	*Operation*

4.3 Extraction of True Occurrences

In order to compute the accuracy (i.e., precision and recall) of a differencing technique we need to determine first the actual changes that occurred between different versions of the examined artifact and consider them as the set of true occurrences. Within the context of object-oriented design differencing we consider the following types of design changes per design aspect.
For *containment*:

- Addition/deletion of an operation or an attribute.
- Change of an operation, which includes any kind of change in its signature (i.e., change of visibility, addition/deletion of modifiers, change of return type, renaming of the operation's name, change in the order of parameters, change in the types of parameters and addition/deletion of parameters).
- Change of an attribute, which includes change of the attribute's visibility, addition/deletion of modifiers, change of the attribute's type and renaming of the attribute's name.

For *inheritance*:

- Addition/deletion/change of the class being extended by a given class.
- Addition/deletion of an interface being implemented by a given class.

For *usage*:

- Addition/deletion of an operation call or attribute access within the body of an operation.
- Change of an operation call. This type of change refers to operation calls which either correspond to operation declarations whose signature has changed or have been replaced with calls to other operations (possibly declared in a different class) that return the same type and possibly take the same arguments as input.
- Change of an attribute access. This type of change refers to attribute accesses which either correspond to changed attribute declarations or have been replaced with accesses to other attributes (possibly declared in a different class) having the same type.

For the extraction of true occurrences we have followed a procedure that ensures, to a large extent, a reliable and unbiased comparison of the examined differencing approaches. Two of the authors of the paper have independently compared the source code of all JFreeChart classes throughout successive stable versions[2].

The comparison has been performed with the help of a source-code differencing tool offered by the Eclipse IDE. The employed tool provides a more sophisticated view of the performed changes in the sense that it is able to associate a change with the context of the source code element where the change occurred. In contrast to traditional text differencing tools, the Eclipse differencing tool offers an additional view as the one illustrated in Figure 3 showing the changes that were performed in class `PaintItem` between versions 1.0.5 and 1.0.6.

In this view, the listed class members are those on which changes have been performed between the two compared versions. Furthermore, the plus (+) and minus (-) symbols indicate that a change occurred in the signature of the corresponding class member (plus symbol is used to represent the previous value of the changed class

[2] `http://sourceforge.net/projects/jfreechart/`

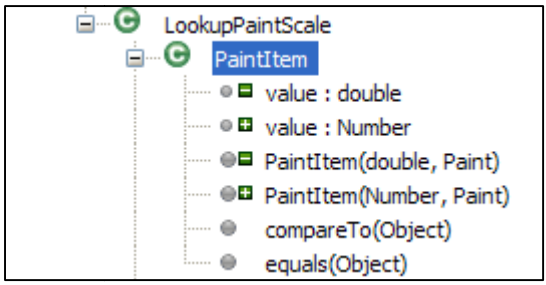


Fig. 3. Differencing view offered by the Eclipse IDE

member, while minus symbol is used to represent the next value of the changed class member). The absence of a symbol indicates that the change occurred within the body of the corresponding class member, thus not affecting its signature. By double-clicking on the elements shown in the differencing view, it is possible to directly inspect the changes on the actual source code and make a safer conclusion about the nature of each change. This differencing view feature offered by the Eclipse IDE made significantly easier, faster and more accurate the manual inspection of the changes that occurred throughout the evolution of JFreeChart. Clearly, this type of inspection is prohibitively time consuming, which is why automated differencing methods are being developed.

The two authors examined 14 successive version pairs for containment and inheritance (ranging from version 1.0.0 to version 1.0.13) and 8 successive version pairs for usage (ranging from version 1.0.0 to version 1.0.8). The reason for selecting a smaller number of version pairs for usage is that the number of usage changes per version pair is significantly larger, thus making the examination of all version pairs impossible. Furthermore, it is significantly harder to manually inspect usage changes, since they occur within the body of the operations, which in turn may have a complex structure and a large number of overlapping changes. Finally, the reason for selecting this specific version range is that the classes in versions prior to 1.0.0 are placed in a completely different package structure making difficult their mapping to the new package structure (introduced after version 1.0.0).[3] Moreover, we have selected the latest versions in the evolution of JFreeChart (until the last/current version 1.0.13), since they cover a more mature development phase of the examined project. Furthermore, they contain a larger number of larger classes, which allows us to test the scalability of the examined differencing techniques.

After the completion of the independent comparison of all classes throughout the aforementioned versions, the two authors merged their results by reaching a common consensus in the cases of a different change interpretation. The cases that required a more careful re-examination usually involved operations or attributes that have been actually renamed. In some of these cases, one of the authors interpreted the change as a deletion of a class member and an addition of a new one, while the other author interpreted it as a change to the same class member.

The number of true occurrences for each type of change per design aspect (i.e., containment, inheritance and usage) is shown in Tables 6, 7, 8 respectively. As it can be

[3] Note that *UMLDiff* can handle this type of overall source-code reorganization, as it is capable of recognizing class moves across packages. *VTracker* is also, in principle, capable, however the time complexity of comparing whole system structures as a trees is prohibitive.

observed from the tables most of the actually performed changes are additions, especially in containment and inheritance aspects. This is not surprising, since JFreeChart is a Java library that is used by client applications for creating and displaying charts. Consequently, its developers tried to maintain a consistent public interface throughout its evolution without performing several deletions and signature changes.

Table 6. True Occurrences for containment (operations and attributes)

Versions	Added oper.	Removed oper.	Changed oper.	Added attr.	Removed atrr.	Changed attr.
1.0.0-1.0.1	10	0	0	1	0	0
1.0.1-1.0.2	60	0	0	17	1	0
1.0.2-1.0.3	86	3	2	29	0	16
1.0.3-1.0.4	70	1	3	9	1	0
1.0.4-1.0.5	85	0	5	11	1	1
1.0.5-1.0.6	78	7	2	22	1	2
1.0.6-1.0.7	125	0	3	50	3	2
1.0.7-1.0.8	36	0	0	6	0	0
1.0.8-1.0.8a	4	0	0	0	0	0
1.0.8a-1.0.9	15	1	1	0	0	0
1.0.9-1.0.10	94	0	3	11	0	6
1.0.10-1.0.11	117	0	1	41	4	3
1.0.11-1.0.12	45	2	0	11	1	4
1.0.12-1.0.13	160	4	6	50	2	0
TOTAL	**985**	**18**	**26**	**258**	**14**	**34**

Table 7. True Occurrences for inheritance (generalizations and realizations)

Versions	Added gener.	Removed gener.	Changed gener.	Added realiz.	Removed realiz.
1.0.0-1.0.1	1	0	0	2	0
1.0.1-1.0.2	3	0	0	3	0
1.0.2-1.0.3	16	0	0	23	0
1.0.3-1.0.4	5	0	0	17	1
1.0.4-1.0.5	3	0	0	5	0
1.0.5-1.0.6	6	0	0	11	0
1.0.6-1.0.7	18	0	0	52	0
1.0.7-1.0.8	0	0	0	0	0
1.0.8-1.0.8a	0	0	0	0	0
1.0.8a-1.0.9	0	0	0	0	0
1.0.9-1.0.10	4	0	0	35	18
1.0.10-1.0.11	6	0	0	23	0
1.0.11-1.0.12	0	0	0	0	18
1.0.12-1.0.13	9	0	0	30	0
TOTAL	**71**	**0**	**0**	**201**	**37**

Table 8. True Occurrences for usage (operation calls and attribute accesses)

Versions	Added oper. calls	Removed oper. calls	Changed oper. calls	Added attr. accesses	Removed atrr. accesses	Changed attr. accesses
1.0.1-1.0.2	119	31	25	51	6	0
1.0.2-1.0.3	306	99	47	72	31	134
1.0.3-1.0.4	180	23	18	82	15	0
1.0.4-1.0.5	143	102	64	109	14	11
1.0.5-1.0.6	266	97	85	36	20	5
1.0.6-1.0.7	210	74	46	106	28	13
1.0.7-1.0.8	84	223	115	21	2	0
TOTAL	**1324**	**650**	**400**	**489**	**117**	**164**

4.4 Evaluation of Precision and Recall

In order to evaluate the accuracy of the two examined differencing approaches, we should compare the set of true occurrences with the results reported by each tool. For this purpose, we have defined a common report format per design aspect (i.e., containment, inheritance and usage) in order to make easier the comparison of the results reported by each tool with the set of true occurrences. Next, we generated human readable textual descriptions of the true occurrences for each examined version pair of JFreeChart and per design aspect (based on the common report format). Finally, we transformed the output produced by each tool to the common report format. In particular, we have created a parser that goes through the changes reported in the edit scripts produced by *VTracker* and generates a report per design aspect following the common format rules. Additionally, we executed a set of appropriate queries on the database tables where *UMLDiff* stores the change facts of interest and transformed the results of the queries into the common report format.

The source code required for the replication of the experiment along with the gold standard containing the actual changes that occurred between the successive versions of JFreeChart and the edit scripts produced by *VTracker* and *UMLDiff* are available online[4].

For the computation of precision and recall we need to define and quantify three measures, namely:

- *True Positives* (TP): the number of true occurrences reported by each examined tool.
- *False Positives* (FP): the number of false occurrences reported by each examined tool.
- *False Negatives* (FN): the number of true occurrences not reported by each examined tool.

After determining the values for the three aforementioned measures the accuracy of each examined tool can be computed based on the following formulas:

[4] `http://hypatia.cs.ualberta.ca/~vtracker/`

$$\text{Precision} = \frac{TP}{TP + FP} \quad (1) \qquad \text{Recall} = \frac{TP}{TP + FN} \quad (2)$$

In Tables 9, 10, 11 we present the results of precision and recall for the containment, inheritance and usage design aspects, respectively.

Table 9. Precision (P) and recall (R) per type of change for containment

	VTracker		*UMLDiff*	
	P (%)	R (%)	P (%)	R (%)
Added operations	100	100	99.4	97.6
Removed operations	100	100	55.5	83.3
Changed operations	100	100	100	100
Added attributes	98.4	98	98.4	98
Removed attributes	75	64.3	64.7	78.6
Changed attributes	83.3	88.2	91.9	100

Table 10. Precision (P) and recall (R) per type of change for inheritance

	VTracker		*UMLDiff*	
	P (%)	R (%)	P (%)	R (%)
Added generalizations	100	100	100	100
Removed generalizations	N/A	N/A	N/A	N/A
Changed generalizations	N/A	N/A	N/A	N/A
Added realizations	100	100	84.4	100
Removed realizations	100	100	N/A	0

N/A: not applicable due to zero by zero division.

Table 11. Precision (P) and recall (R) per type of change for usage

	VTracker		*UMLDiff*	
	P (%)	R (%)	P (%)	R (%)
Added operation calls	99	93.6	83.6	87.7
Removed operation calls	99.3	88.5	99.6	92
Changed operation calls	79.7	100	100	82.2
Added attribute accesses	99.8	97.1	98.5	95.3
Removed attribute accesses	99	88	98.9	77.8
Changed attribute accesses	92.1	100	100	6.1

4.4.1 VTracker

As shown in Table 9, *VTracker* demonstrated an absolute precision and recall in identifying the actual changes that occurred in operations, but failed to identify correctly some changes which were related to attributes. In total, *VTracker* missed 4 changes in attributes:

- In versions 1.0.4-1.0.5 and class `AbstractBlock`, the attribute `border` with type `BlockBorder` was changed to attribute `frame` with type `BlockFrame`. This double change (i.e., attribute renaming and type change) was reported as a removal of attribute `border` from version 1.0.4 and an addition of attribute `frame` in version 1.0.5.
- In versions 1.0.10-1.0.11 and class `XYDrawableAnnotation`, the attributes `width` and `height` were renamed to `displayWidth` and `displayHeight`. *VTracker* produced an incorrect mapping of the renamed attributes with other attributes of the class.
- In versions 1.0.10-1.0.11 and class `PaintScaleLegend`, the static and final attribute `SUBDIVISIONS` was changed to non-static and non-final attribute `subdivisions`. This change was reported as a removal of the original attribute and an addition of a new one.

Moreover, *VTracker* reported erroneously 6 cases of attribute changes that were actually removals of fields from previous versions and additions of new ones.

Table 10 shows that *VTracker* demonstrated an absolute precision and recall in identifying inheritance related changes.

Finally, *VTracker* demonstrated a relatively high precision and recall in identifying usage-related changes (see Table 11). The lowest percentage is observed in the precision for changed operation calls (79.7%). This is due to a significant number of cases that were identified as changed operation calls, while actually they correspond to removals of operation calls from previous versions (usually by deleting code fragments within the body the operations) and additions of new operation calls.

4.4.2 UMLDiff

In general, *UMLDiff* demonstrated a high precision and recall in identifying containment related changes (Table 9). In comparison with *VTracker*, *UMLDiff* performed better in the identification of changed attributes. This means that the use of domain-specific heuristics (e.g., by combining attribute usage information) can lead to better results especially with respect to the renaming of attributes.

As shown in Table 10, *UMLDiff* failed to identify correctly all removals of realizations. Moreover, the realizations that were supposed to be reported as removed were actually reported as added (false positives). As a result, this situation had also a negative impact on the precision of added realizations. All problematic cases refer to subclasses that implemented a list of interfaces in a previous version that were removed in the next version. However, the same list of interfaces was implemented by their superclasses in both previous and next versions. We believe that this inaccuracy is caused by the fact that *UMLDiff* computes and reports transitively all inheritance relationships (i.e., the generalizations and realization relationships of a superclass are also considered as direct relationships for all of its subclasses).

Regarding usage-related changes, *UMLDiff* demonstrated a low recall in identifying changed attribute accesses (6.1%). All problematic cases refer to accesses of attributes that were renamed or whose type has changed between two versions. Possibly, *UMLDiff* considers that the access itself does not change when the attribute that it refers to is changed.

4.4.3 Comparison of Overall Accuracy

In Table 12 we present the overall precision and recall (i.e., over all types of changes) per design aspect. It is obvious that *VTracker* demonstrated better overall precision and recall in all examined design aspects. This result can be mainly attributed to the fact that *VTracker* performed better on the changes related to operations and operation calls (especially to the operations and operation calls that have been added, Table 9 and 11) whose number is significantly larger compared to the other types of changes (Table 6 and 8) and thus its overall precision and recall was positively affected.

Table 12. Overall precision and recall

	VTracker		*UMLDiff*	
	P (%)	R (%)	P (%)	R (%)
Containment	99	98.9	97.7	97.4
Inheritance	100	100	88.1	88.1
Usage	95.6	94	91.8	84.4

It is very important to note that the improved accuracy in the results of *VTracker* was achieved by using the default implementation of the tree-differencing algorithm and without performing any kind of tuning in the default comparator or similarity function. As already explained in Sections 5.1 and 5.2, we used *VTracker* "out of the box" (so to speak) simply defining the XML input format for each examined design aspect and specifying the required configuration options. The obtained experimental results on the identification of design changes in object-oriented models open the way for the application of *VTracker* (and possibly other domain-independent differencing approaches) on other software engineering differencing problems whose artifacts can be represented in the form of XML.

4.5 Evaluation of Efficiency and Scalability

In order to assess the efficiency and scalability of *VTracker*, we have measured the CPU time required in order to compare the set of XML file pairs corresponding to all the classes of JFreeChart in a given version pair. We performed this analysis for all 14 examined version pairs (starting from version 1.0.0 until version 1.0.13) and per design aspect separately. The measurements have been performed on a MacBookPro5,1 (Intel Core 2 Duo 2.4 GHz and 4 GB DDR3 SDRAM). The results of the analysis are shown in Figure 4.

As it can be observed from Figure 4, the inheritance design aspect requires the least amount of CPU time (ranging from 16 to 20 seconds for all the classes in a given version pair), the containment design aspect requires a larger amount of CPU time (ranging from 300 to 458 seconds), while the usage design aspect requires the largest amount of CPU time (ranging from 3843 to 6292 seconds, approximately 64 to 105 minutes). From a more detailed analysis of the results, we can conclude that there is an almost linear relation between the size of the compared trees (in terms of the number of their nodes) and the time required for their comparison. For example, when the size of the compared trees is increased by 10 times, the time required for their

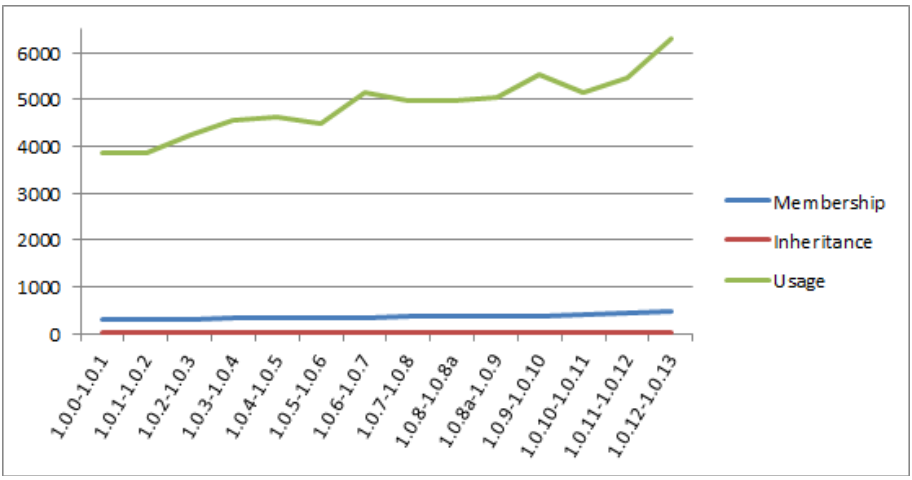


Fig. 4. CPU time (in seconds) per examined version pair and design aspect for *VTracker*

comparison in also increased by 10 times. This outcome may initially not seem intuitive, since the problem of matching ordered labeled trees is quadratic to the number of nodes by nature. However, *VTracker* applies a set of heuristics (described in Section 3) that make the performance of the tree differencing algorithm linear for a major part of the matching problem and quadratic for the rest.

Another interesting observation is that the time required for the analysis of a version pair increases as JFreeChart evolves. This phenomenon can be attributed to two reasons: first because the number of classes increased as the project evolved, and second because the size of some classes increased as the project evolved.

Additionally, we have measured the CPU time required by *UMLDiff* for the comparison of each JFreeChart version pair in order to provide a direct comparison of efficiency between the two differencing approaches. Figure 5 shows the CPU time required for the comparison of each JFreeChart version pair by *VTracker* and *UMLDiff*, respectively. In the case of *VTracker*, the given CPU time is actually the sum of the CPU times required for differencing each design aspect (Figure 4).

As it can be observed from Figure 5, *UMLDiff* performed better in every examined JFreeChart version pair and required on average 27% less CPU time compared to *VTracker*, even though it considered all design aspects in the same context. The separation of the three design aspects is necessary to make the use of *VTracker* feasible for large systems (otherwise it suffers from insufficient-memory problems and fails). This simplification of the problem also has a positive impact to the accuracy of *VTracker,* which is difficult to quantify however. From *VTracker*'s efficiency analysis per design aspect, we estimated that the comparison of the XML files representing usage constitutes 93% of the total CPU time. The XML files for the usage design aspect have exactly the same structure as the XML files for containment (Figure 2) with the addition of nodes representing operation calls and attribute accesses (as children of the `Operation` nodes). As a result, the XML files for the usage aspect contain a significantly larger number of nodes and their alignment

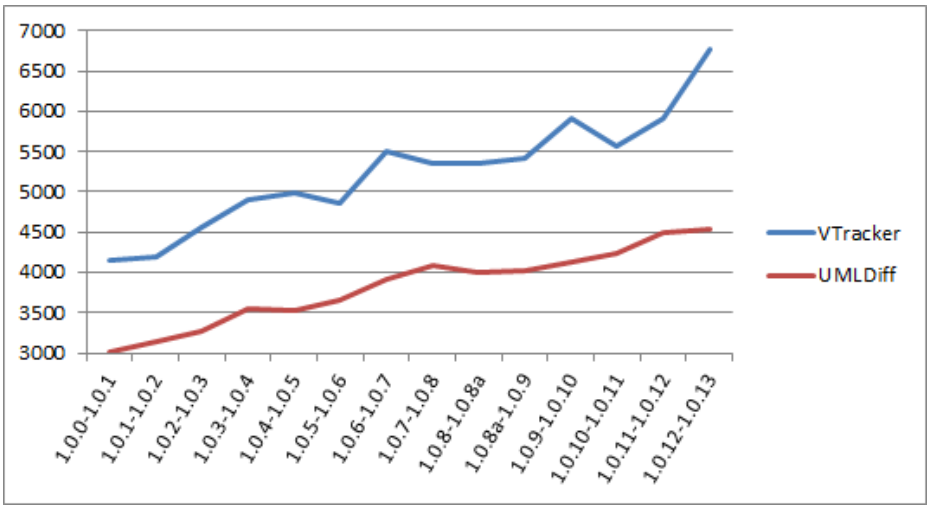


Fig. 5. CPU time (in seconds) per examined version pair for *VTracker* and *UMLDiff*

requires significantly more processing time, since matching is performed at two levels (i.e., the `Operation` level and the `OperationCall` and `AttributeAccess` level). However, the fact that *VTracker* can analyze each design aspect separately makes it a more efficient solution for the detection of API-level changes (i.e., changes in the public interface of the examined classes that can be detected by analyzing the containment and inheritance design aspects).

4.6 Threats to Validity

Let us now consider the various threats to the validity of our experiment and findings. In principle, the internal validity of our experiment could potentially be threatened by erroneous application of tools and incorrect observations and interpretations by the experimenters. On the other hand, the threats to external validity of the conducted experiment are associated with factors that could limit the generalization of the results to other examined projects, differencing algorithms and domains.

4.6.1 Internal Validity

The first threat to the internal validity of the conducted experiment is related with the determination of true occurrences. Obviously, the extracted set of true occurrences affects the computation of both precision and recall and consequently could also affect the conclusions of the experiment. This threat was alleviated by two means. First, the extraction of design changes was performed independently by two of the authors and their results were merged by reaching a common consensus in the cases of a different change interpretation. In this way, we tried to eliminate the bias in the interpretation of changes. Second, the authors inspected the changes with the help of a sophisticated source code differencing tool offered by the Eclipse IDE. This tool

made easier and more accurate the inspection and interpretation of changes in comparison to generic text differencing tools which are not able to associate a change with the context of the source code element where the change occurred. In this way, we tried to eliminate human errors in the process of manually identifying source code changes.

The second threat to the internal validity of the conducted experiment is related with the correct and proper use of the examined differencing tools. Obviously, this could affect the results being reported by the examined tools and consequently the conclusions of the experiment. This threat was alleviated by taking advice directly from the developers of the tools (who are also authors of this paper) on how to properly configure, execute and collect the change information. More specifically, the developer of *UMLDiff* (Xing) specified the queries required for the extraction of the examined design changes from the database in which the change facts are stored. Furthermore, the developer of *VTracker* (Mikhaiel) gave advice towards the construction of XML input files that optimize the accuracy and efficiency of *VTracker*, the proper configuration of *VTracker* for the employed XML schema representation, and finally the correct parsing of the produced edit script describing the changes.

4.6.2 External Validity

Regarding the generalization of the results to other projects, we have selected an open-source project, namely JFreeChart, which has been widely used as a case study in several empirical studies and source code differencing experiments in particular. Therefore, it can be considered as a rather representative and suitable project for this kind of experiments. However, it should be noted that JFreeChart is a project that evolved mostly by adding new features and fixing bugs. Moreover, due to the fact that it is a library, it has not been subject to a large number of refactoring activities (a heavily refactored library would cause several compilation problems to already existing client applications). Obviously, the presence of complicated refactorings in the evolution of a project would have a significant impact on the accuracy of any differencing technique. As a result, we cannot claim that the results can be generalized to any kind of software projects (e.g., frameworks, APIs, applications).

Regarding the generalization of the results to other differencing algorithms, we have compared a generic domain-agnostic algorithm (*VTracker*) with a domain-specific algorithm (*UMLDiff*), which is considered as the state-of-the-art in the domain of object-oriented model differencing. Several prior experimental studies [19], [25] have demonstrated a high accuracy for *UMLDiff* in accordance with the results of this experiment. Therefore, it can be considered as one of the best differencing algorithms in its domain.

Finally, regarding the generalization of the results to other domains, we have selected a domain, namely object-oriented design models, which is very rich in terms of model elements and relationships among them. As a result, we could assume that our generic algorithm would demonstrate a similar performance in domains having a similar or lower complexity, such as Web service specification documents in the form of WSDL files. However, this assumption needs to be empirically validated with further experiments.

5 Related Work

The general area of software-model differencing is quite vast. A pretty comprehensive overview can be found in Chapter 2 of Xing's thesis [28]. In this paper, we eclectically review the most relevant work (Section 5.1) and we discuss the work of our own team building on *UMLDiff* and *VTracker* (Section 5.2).

5.1 Object-Oriented Design Differencing

Object-oriented software systems are better understood in terms of structural and behavioral models, such as UML class and sequence models. The UML modeling tools often store UML models in XMI (XML Metadata Interchange) format for data-interchange purposes. XML-differencing tools (such as DeltaXML[5] for example), applied to these easily available XMI representations, report changes of XML elements and attributes, ignoring the domain-specific semantics of the concepts represented by these elements. *VTracker*, with its domain-aware affine cost function and its ability to take into account references, is exactly addressing this problem of domain-aware XML differencing. *VTracker* (and its precursor algorithms) has in fact been applied to other domains, including HTML comparison [9], RNA alignment [7], and WSDL comparison [8, 31].

In the context of UML differencing, several UML modeling tools come with their own UML-differencing methods [2, 11]. Each of these tools detect differences between subsequent versions of UML models, assuming that these models are manipulated exclusively through the tool in question which manages persistent identifiers for all model elements. Relying on consistent and persistent identifiers is clearly not possible if the development team uses a variety of tools, which is usually the case.

More generally, on the subject of reasoning about similarities and differences between UML models, we should mention Egyed's work [3] on a suite of rule- and constraint- and transformation-based methods for checking the consistency of the evolving UML diagrams of a software system. Similarly, Selonen et al. [13] have also developed a method for UML transformations, including differencing.

Kim et al. [5] developed a method for object-oriented software differencing that works at the level of the source code itself (and does not require its design model). The algorithm takes as an input two versions of a program and starts by comparing the method headers from each program version and identifying the ones that most match at the lexical level, based on a set of matching rules and a similarity threshold. The algorithm iteratively and greedily selects the best rule to apply to identify the next pair of matching methods in order to maximize the total number of matches. This idea was later extended to LSDiff (Logical Structural Diff) [4], which involves more rules.

More recently, Xing [29] proposed a general framework, GenericDiff, for model comparison. GenericDiff represents a domain-independent approach for model

[5] Mosell EDM Ltd: `http://www.deltaxml.com`

differencing that is also aware of domain-specific properties and syntax. In this approach the domain-specific inputs are separated from the general graph matching process and are encoded by using composite numeric vectors and a pair-up graph. This allows the domain-specific properties and syntax to be uniformly handled during the matching process. GenericDiff is similar to *VTracker*, in that they both model the subject systems in terms of a more abstract representation; they are different in that GenericDiff adopts a bipartite-graph model where *VTracker* adopts a tree model.

5.2 Work Building on UMLDiff and VTracker

In this section, we review research from our team, building on *UMLDiff* and *VTracker* for different use cases in design differencing: (a) understanding the ***design changes*** between two versions of a system; (b) analyzing the ***evolution history*** of a system and its constituent components; (c) comparing the ***intended vs. the as-implemented*** design of a system; and (d) ***merging*** out-of-sync versions of software.

Both *UMLDiff* and *VTracker* have been applied to the task of UML-design differencing. *UMLDiff* was implemented in the context of JDEvAn [27], an Eclipse plugin, which can be invoked by the developer to query a pre-computed database of design changes and the analyses based on them. The envisioned usage of *UMLDiff* in the context of JDEvAn was that it would be applied as an off-line process to pairs of "stable" releases of the system as a whole and its results would be made available to developers in the context of their development tasks, i.e., looking at the recent changes of an individual class, or reviewing the refactorings across the system during the most recent releases.

VTracker, on the other hand, was implemented as a service accessible through WebDiff [14], a web-based user interface. In the context of the WebDiff portal, *VTracker* can be applied to any level of logical models, including models of systems, packages or individual classes. Table 13 below identifies the publications in which these studies are described in detail.

Table 13. Studies with UMLDiff and VTracker

	UMLDiff/JDEvAn	**VTracker/WebDiff**
design changes	**19, 25**	**14**
longitudinal class/system analysis	**16, 17, 18, 21, 23**	
design vs. code differencing		**14**
refactoring and merging	**22, 24, 26**	

5.2.1 Longitudinal Analysis of Individual Classes and the Overall System

Ever since Lehman and Belady first formulated the "Laws of Software Evolution" in 1974, describing the balance between forces driving new software development and forces that slow down progress and increase the brittleness of a system, software-engineering research has been investigating different metrics and methods for analyzing evolution to recognize the specific forces at play at a particular point in the life of a system.

Relying on *UMLDiff*, we developed a method for analyzing the long-term evolution history of a system as a whole, its individual classes, and related class collections, based on metrics summarizing its design-level changes. Given a sequence of UML class models, extracted from a corresponding sequence of code releases, we can use *UMLDiff* to extract the design-level changes between each pair of subsequent code releases, to construct a sequence of system-wide *system-change transactions* and class-specific *class-change transactions*.

To analyze potential co-evolution patterns between sets of system classes [18, 23], we first discretized the class-change transactions into a sequence of 0s (when there was no change to the class) and 1s (if there was at least some change to the class). In a subsequent experiment, we conducted a more refined discretization process, classifying the collection of changes that each class suffered into one of five discrete categories, depending on whether they have high/low/average number of element additions/deletions/changes. We then applied the Apriori association-rule mining algorithm to recognize sets of coevolving classes (as itemsets). Recognizing coevolving classes is interesting since co-evolution implies design dependencies among the coevolving classes; when such dependencies are undocumented, they are likely to be unintentional and possibly undesirable. In fact co-evolution is frequently referred to as a "bad design smell" implying the need for refactoring.

In addition to co-evolution, we have explored two more types of analyses of longitudinal design evolution. We used phasic analysis to recognize distinct phases in the discretized evolution profile of a design entity, whether it is the system as a whole or an individual class. Intuitively, a phase consists of a consecutive sequence of system versions, all of which exhibit similar classifications of changes. Identifying a phase in a class-evolution profile may provide some insight regarding the development goals during the corresponding period. We further used Gamma analysis to recognize recurring patterns in the relative order of phases in an evolution profile, such as consistent precedence of a phase type over another. Different process models advocate distinctive ordering of activities in the project lifecycle; gamma analysis can reveal such consistent relative orderings and, thus, hint at the adopted process model. In particular, Gamma analysis provides a measure of the general order of elements in a sequence and a measure of the distinctiveness or overlap of element types.

Finally, we developed a set of special-purpose queries [22, 24] to the design-changes database to extract information about combination of design-level changes characteristic of refactorings.

5.2.2 Design vs. Code Differencing

We have experimented with reflexion, i.e., comparison between design (as intended) vs. design as implemented in the code (extracted through reverse-engineering tools) using the *VTracker* through the WebDiff portal. It is interesting to note here that although both *UMLDiff* and *VTracker* are equally applicable (and able to address) to this task, pragmatically *VTracker* is a better choice. Since *UMLDiff* is implemented as a java-based program accessing a database of extracted design-level facts, to apply it to this task, we would have to develop a parser for XMI to extract the relevant design facts from a UML design and store them in the JDEvAn [27] database for *UMLDiff*.

VTracker, on the other hand, requires as input XML documents easily available as the products of either a design tool or a reverse engineering tool.

5.2.3 Software Merging

A particularly interesting case of software merging is that of migrating applications to newer versions of libraries and/or frameworks. Applications built on reusable component frameworks are subject to two independent, and potentially conflicting, evolution processes. The application evolves in response to the specific requirements and desired qualities of the application's stakeholders. On the other hand, the evolution of the component framework is driven by the need to improve the framework functionality and quality while maintaining its generality. Thus, changes to the component framework frequently change its API on which its client applications rely and, as a result, these applications break.

Relying on *UMLDiff*, in the Diff-CatchUp tool [26], we tackled the API-evolution problem in the context of reuse-based software development, which automatically recognizes the API changes of the reused framework and proposes plausible replacements to the "obsolete" API based on working examples of the framework code base. The fundamental intuition behind this work is that when a new version of the framework is developed, it is usually associated with a test suite that exercises it. This test suite constitutes an example of how to use the new framework version and can be used as an example for other client applications that need to migrate.

6 Summary and Conclusion

In this paper, we reviewed two different algorithms and their corresponding tool implementations for object-oriented design differencing, a task that is essential for the purposes of (a) recognizing design-level changes between two versions of a software system; (b) comparing the intended design of a system against its as-implemented design; (c) analyzing the long-term evolution of a system and its constituent components; and (d) merging out-of-sync versions of software.

UMLDiff and *VTracker* assume the same basic conceptual model of UML models, namely, as trees, where nodes correspond to design elements, their children correspond to the elements' contents, and additional edges connect them to other "related" design elements. The actual representations on which the two algorithms operate are different. *UMLDiff* works on a database of design facts, precisely reflecting the UML relations in the system. *VTracker* works on XML documents and primarily exploits and relies on the tree structure of these documents, as opposed to the semantics of the underlying UML relations they represent. Together, they give us an interesting test-bed on which to study software-model differencing in general.

In order to compare the two approaches, we first extracted the actual design changes that occurred between successive versions of the JFreeChart open-source project and used them as the set of true occurrences. This gold standard has been made publicly available and can serve as a benchmark for the evaluation of other differencing techniques, as well as for the replication of the conducted experiment. Based on the extracted set of true occurrences we computed the precision and recall of *VTracker* and *UMLDiff* and compared their accuracy for several types of changes

within three design aspects, namely containment, inheritance and usage. In general, *VTracker* proved to be more accurate than *UMLDiff* over most types of changes per design aspect despite of being domain-independent. *UMLDiff* performed better than *VTracker* only in the identification of changed attributes. The experimental results open the way for the application of *VTracker* on other software engineering differencing problems whose artifacts can be represented in the form of XML.

Finally, we performed an efficiency analysis based on the CPU time required by *VTracker* and *UMLDiff* for the comparison of all classes per version pair of JFreeChart. We concluded that *VTracker* has a comparable performance to *UMLDiff*, since *VTracker* required on average 27% more CPU time compared to *UMLDiff*. Additionally, the analysis has shown that there is an almost linear relation between the size of the compared trees (in terms of the number of their nodes) and the time required for their comparison and thus the *VTracker* algorithm can be efficiently applied to domains of problems having even a larger size.

The fundamental contribution of this study is that it demonstrates *VTracker*'s relevance to software difference, as a flexible and effective tool for recognizing changes in software evolution. In the future, we plan to apply *VTracker* to more instances of this general problem, by developing more XML representations of software, towards producing a general software differencing service.

Acknowledgements. Many more people have been involved in this work over the years during which these algorithms were being developed and evaluated, including Brendan Tansey, Ken Bauer, Marios Fokaefs and Fabio Rocha. Their contributions towards this body of work have been invaluable and we are grateful for them. This work has been supported by NSERC, AITF (former iCORE) and IBM.

References

1. Dulucq, S., Tichit, L.: RNA Secondary structure comparison: exact analysis of the Zhang–Shasha tree edit algorithm. Journal Theoretical Computer Science 306(13), 471–484 (2003)
2. Comparing and merging UML models in IBM Rational Software Architect, `http://www-128.ibm.com/developerworks/rational/library/05/712_comp/`
3. Egyed, A.: Scalable consistency checking between diagrams - The VIEWINTEGRA approach. In: Proceedings of the 16th International Conference on Automated Software Engineering, pp. 387–390 (2001)
4. Kim, M., Notkin, D.: Discovering and Representing Systematic Code Changes. In: Proceedings of the 31st International Conference on Software Engineering, pp. 309–319 (2009)
5. Kim, M., Notkin, D., Grossman, D.: Automatic Inference of Structural Changes for Matching Across Program Versions. In: Proceedings of the 29th International Conference on Software Engineering, pp. 333–343 (2007)
6. Levenshtein, V.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)

7. Mikhaiel, R., Lin, G., Stroulia, E.: Simplicity in RNA Secondary Structure Alignment: Towards biologically plausible alignments. In: Post Proceedings of the IEEE 6th Symposium on Bioinformatics and Bioengineering, pp. 149–158 (2006)
8. Mikhaiel, R., Stroulia, E.: Examining Usage Protocols for Service Discovery. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 496–502. Springer, Heidelberg (2006)
9. Mikhaiel, R., Stroulia, E.: Accurate and Efficient HTML Differencing. In: Proceedings of the 13th International Workshop on Software Technology and Engineering Practice, pp. 163–172 (2005)
10. Mikhaiel, R.: Comparing XML Documents as Reference-aware Labeled Ordered Trees, PhD Thesis, Computing Science Department, University of Alberta (2011)
11. Ohst, D., Welle, M., Kelter, U.: Difference tools for analysis and design documents. In: Proceedings of the 19th International Conference on Software Maintenance, pp. 13–22 (2003)
12. Schofield, C., Tansey, B., Xing, Z., Stroulia, E.: Digging the Development Dust for Refactorings. In: Proceedings of the 14th International Conference on Program Comprehension, pp. 23–34 (2006)
13. Selonen, P., Koskimies, K., Sakkinen, M.: Transformations between UML diagrams. Journal of Database Management 14(3), 37–55 (2003)
14. Tsantalis, N., Negara, N., Stroulia, E.: WebDiff: A Generic Differencing Service for Software Artifacts. In: Proceedings of the 27th IEEE International Conference on Software Maintenance, pp. 586–589 (2011)
15. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of the ACM 21(1), 168–173 (1974)
16. Xing, Z., Stroulia, E.: Understanding Phases and Styles of Object-Oriented Systems' Evolution. In: Proceedings of the 20th International Conference on Software Maintenance, pp. 242–251 (2004)
17. Xing, Z., Stroulia, E.: Understanding Class Evolution in Object-Oriented Software. In: Proceedings of the 12th International Workshop on Program Comprehension, pp. 34–45 (2004)
18. Xing, Z., Stroulia, E.: Data-mining in Support of Detecting Class Co-evolution. In: Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering, pp. 123–128 (2004)
19. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 54–65 (2005)
20. Xing, Z., Stroulia, E.: Towards Experience-Based Mentoring of Evolutionary Development. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 621–624 (2005)
21. Xing, Z., Stroulia, E.: Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software. IEEE Trans. Software. Eng. 31(10), 850–868 (2005)
22. Xing, Z., Stroulia, E.: Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance, pp. 458–468 (2006)
23. Xing, Z., Stroulia, E.: Understanding the Evolution and Co-evolution of Classes in Object-oriented Systems. International Journal of Software Engineering and Knowledge Engineering 16(1), 23–52 (2006)
24. Xing, Z., Stroulia, E.: Refactoring Detection based on UMLDiff Change-Facts Queries. In: Proceedings of the 13th Working Conference on Reverse Engineering, pp. 263–274 (2006)

25. Xing, Z., Stroulia, E.: Differencing logical UML models. Autom. Softw. Eng. 14(2), 215–259 (2007)
26. Xing, Z., Stroulia, E.: API-Evolution Support with Diff-CatchUp. IEEE Trans. Software Eng. 33(12), 818–836 (2007)
27. Xing, Z., Stroulia, E.: The JDEvAn tool suite in support of object-oriented evolutionary development. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008 Companion), pp. 951–952 (2008)
28. Xing, Z.: Supporting Object-Oriented Evolutionary Development by Design Evolution Analysis, PhD Thesis, Computing Science Department, University of Alberta (2008)
29. Xing, Z.: Model Comparison with GenericDiff. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, pp. 135–138 (2010)
30. Zhang, K., Shasha, D.: Simple fast algorithm for the editing distance between trees and related problems. SIAM Journal on Computing 18(6), 1245–1262 (1989)
31. Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., Lau, A.: An Empirical Study on Web Service Evolution. In: Proceedings of the IEEE International Conference on Web Services, ICWS 2011, pp. 49–56 (2011)