

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Measuring the quality of diff algorithms: A formalization

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:

Barabucci, G., Ciancarini, P., Di Iorio, A., Vitali, F. (2016). Measuring the quality of diff algorithms: A formalization. *COMPUTER STANDARDS & INTERFACES*, 46, 52-65 [10.1016/j.csi.2015.12.005].

Availability:

This version is available at: <https://hdl.handle.net/11585/560957> since: 2016-09-03

Published:

DOI: <http://doi.org/10.1016/j.csi.2015.12.005>

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>).
When citing, please refer to the published version.

(Article begins on next page)

This is the final peer-reviewed accepted manuscript of:

Barabucci, G., et al. "Measuring the Quality of Diff Algorithms: A Formalization." *Computer Standards and Interfaces*, vol. 46, 2016, pp. 52-65.

The final published version is available online at:
<http://dx.doi.org/10.1016/j.csi.2015.12.005>

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>)

When citing, please refer to the published version.

Measuring the quality of diff algorithms: a formalization

Gioele Barabucci^b, Paolo Ciancarini^a, Angelo Di Iorio^c, Fabio Vitali^a

^aUniversity of Bologna, Italy

^bUniversität zu Köln, Germany

^cUniversity of Bologna, Italy - contact author: angelo.diiorio@unibo.it, phone: +390512094898, fax: +390512094510

Abstract

The automatic detection of differences between documents is a very common task in several domains. This paper introduces a formal way to compare diff algorithms and to analyse the deltas they produce. There is no one-fits-all definition for the quality of a delta, because it is strongly related to the application domain and the final use of the detected changes. Researchers have historically focused on minimality: reducing the size of the produced edit scripts and/or taming the computational complexity of the algorithms. Recently they started giving more relevance to the human interpretability of the deltas, designing tools that produce more readable, usable and domain-oriented results. We propose a universal delta model and a set of metrics to characterise and compare effectively deltas produced by different algorithms, in order to highlight what are the most suitable ones for use in a given task and domain.

1. Introduction

The automatic comparison of two different versions of a document and the compilation of a list of changes between them is a common task. A *diff* algorithm is used for this purpose: it takes two files as input and computes their difference, according to a given set of change operations. The outputs of diff algorithms, usually called *deltas*, *diffs* or *patches*, are used for many purposes: programmers review source code diffs to avoid adding bugs and to understand which parts of the code has changed; editors highlight the changes made on drafts and pre-prints; law makers compare proposals during the discussion and approval of a bill; philologists use the differences between documents to recreate the *stemma codicum* of a text, the history of its development. An exhaustive survey on change detection and versioning tools can be found in [1].

Historically the research on diff algorithms has been carried out by the database community, that has to deal with huge quantities of data and seeks to reduce space and time consumption. In fact, these algorithms have been evaluated mainly by comparing their time and space performance. Almost all the experiments in the literature follow the same pattern: the authors first compare the computational complexity and the execution time of the algorithms, then evaluate the quality of the results, see for instance [2][3][4][5].

The quality is often expressed in terms of the ability to reduce the size of the produced delta. As summarized in [6]: “quality is described by some minimality criteria [...] Minimality is important because it captures to some extent the semantics that a human would give when presented with the two versions”.

Surprisingly only a few other quality measures have been defined and applied. There is now a growing interest in characterizing more precisely the quality of deltas, in order to design

November 13, 2015

algorithms that produce an output that is *easier to interpret* and more adequate for human readers, for instance specialized for literary documents [7] or ontological data [8].

The focus of this paper is on comparing the quality of the deltas produced by diff algorithms.

We introduce a framework for measuring the quality of diffs through an objective evaluation process. The basic idea consists of extracting numerical indicators from deltas (such as the number of detected changes, the number of high-level changes, the number of elements listed in the description of each change) and aggregating them into more complex quantitative metrics. These indicators can be associated to quality requirements and evaluated to decide whether or not the algorithm that produced that delta is ‘better’ than others in a given context.

This work is built on top of UniDM [9][10], a unified conceptual model able to abstract the characteristics of deltas. Each diff algorithm uses its own strategy either in computing deltas or in serializing. Design choices are strictly dependent on the application domain and, very often, prescribed by the tools that are meant to apply deltas. Such a unified model, along with the evaluation metrics on top of it, gives users a powerful tool to analyze in a more precise way the behavior of the algorithms. It is also worth remarking that this model is general enough to deal with streams of text, lists, trees or graphs, so that the same evaluation process can be applied to heterogeneous algorithms and domains.

The paper is structured as follows. Section 2 describes in more depth the solutions adopted to evaluate the quality of diff algorithms. Section 3 discusses how the same concept of ‘quality’ can have different meanings according to users’ needs and preferences. Section 4 introduces our solution, that is detailed in each part in the following sections: Section 5 introduces UniDM (the model used to analyse the deltas) and some quantitative indicators, while Sections 6 describes the metrics in formal way. The application of the metrics is presented in Section 7: we introduce a two-phase method to evaluate the existing algorithms and we present experimental results on some well-known XML diff tools, before concluding in Section 8.

2. Related work

It is hard to compare the quality of the output of diff algorithms. First of all, because different algorithms might produce different deltas that are all correct. There is a further tricky issue. As highlighted by [3] “*all approaches make use of different delta models, which makes it difficult to measure the quality of the resulting deltas*”. In fact, each algorithm uses its own internal model and recognizes its own set of changes.

The quality of a delta has been often associated to some notion of minimality; a few alternative characterizations have been proposed.

2.1. Quality as minimality

The most used parameter to measure the quality of diff algorithms is its ability to reduce the dimension of the delta. There are two main approaches to calculate such a dimension: either measuring the size of the file or counting the number of edits listed in the delta. The first approach has been used, for instance, in [11] and [6]. This evaluation can be fully automated and makes it possible to compare directly heterogeneous deltas but is not precise, as it does not investigate the content of the file. The measurement of the number of edits, known as *edit distance*, was experimented in many other works, for instance in the evaluation of DocTreeDiff [3], Xandy [12], X-Diff [4] and XRel_Change_SQL [13].

A refinement of the first approach has been proposed for measuring the quality of Fxma[2]. The authors compared some algorithms by comparing the size of compressed deltas. The motivation is that: “[the authors] expect to get results that are less dependent on the encoding and more closely related to the amount of actual information. The difference in output size due to some tools generating XML and others binary diffs should be mitigated by compression”. This approach reduces the noise generated by implementation choices of each algorithm.

The edit distance approach is more precise, though is heavily influenced by the set of available operations. In fact, this model initially considered insertion and deletion operations only. Later, other edits have also been considered such as the substitution of elements’ labels and names, of intermediate nodes (including their attributes) and of entire subtrees. The introduction of these complex operations required more flexible models for calculating the weights of each operation. The minimization of the edit distance was further refined by deploying *edit cost models*. This is the solution proposed in the early days of tree-based diff algorithms by [14]. The idea is to pre-define a cost for each type of change and to measure the overall cost of the delta as the sum of the costs of each detected change. In the same paper the authors introduce an algorithm that minimizes this cost.

2.2. Quality as interpretability

In other cases, the quality of a delta has been associated to the capability of humans to interpret and exploit the changes it contains. This quality is much more difficult to define, as it involves the nature of changes and the human analysis of the output.

For instance, in [7] the authors introduced the notion of *naturalness* of a diff algorithm. The naturalness indicates the “*capability of producing an edit script that an author would recognize as containing the changes she/he effectively performed when editing a document*”. The authors presented a taxonomy of natural operations on literary documents and an algorithm, called JN-Diff, able to capture (most of) those operations. The focus is on the quality of deltas in terms of readability and accuracy for human users, so that JNDiff is slower than other algorithms, but still acceptable.

The idea of looking for deltas that better describe operations on literary documents has also been investigated by DocTreeDiff [3]. In the same paper, the authors sketched out an original approach to measure quality. They suggested to compare the mixture of changes listed in the deltas. The analysis is quite preliminary but shows a great variety of the types of changes detected by the algorithms. The ability of an algorithm to detect a larger and more precise set of changes is considered a good indicator of quality.

The identification of higher level changes that capture appropriately the editing process has also been studied for XML database schema evolution [15]. The authors discuss a taxonomy of high-level changes and how each change can be expressed as combination of smaller units. As stated in the paper, however, their model is incomplete and does not cover all possible XML DTDs and schemas.

Other interesting ways of assessing the quality of deltas have been proposed in the area of ontology diffing. In [16] the authors discussed the need of a high-level set of changes that should be detected in order to produce deltas that are “*more intuitive, concise, closer to the intentions of the ontology editors*” and that “*capture more accurately the semantics of changes*”. They proposed both a set of high-level changes, described in a formal way, and an algorithm to detect them. From authors’ perspective, in fact, the presence of high-level changes in the delta increases its quality and effectiveness. In [8] the authors measured the quality of the deltas between ontologies as a combination of heterogeneous properties such as reversibility, size minimality and

redundancy elimination. The authors introduced multiple differential functions to compute deltas and argued that the quality depends on types of information extraction and reasoning that are expected on changes.

The importance of letting users to tune the quality of a diff algorithm in relation to the set of detectable high-level changes was also stressed by [17]. The authors, in fact, introduced the idea of *viewpoints* (i.e., each ontology designer has her/his own needs and should be able to define the set of complex changes she/he is interested in) and proposed a language to describe complex changes, called CDL (Change Definition Language).

We agree that measuring the quality of a delta as a single fixed value is not enough. A better approach is to think of deltas as objects that have multiple measurable dimensions, each able to capture one facet of quality. There are in fact contrasting needs and expectations in characterising such dimension.

3. Quality of deltas in different scenarios

Users in different domains have different requests and expectations on deltas and their quality. In this section we present some application scenarios and we explain how a more precise characterization of the quality of (the output of) diff algorithms would help users in selecting the most suitable solution for their purposes. The scenarios are identified with labels S1, S2, ..., Sn that will be used throughout the paper.

The diff algorithms are widely used by programmers. Different programmers, or even the same programmer in different moments, might have different needs. Consider, for instance, the following two scenarios:

S1: Programmer developing code: some verbosity is appreciated by developers who review source code during the development. It is often required to have contextual information that wraps the actual changes detected by a diff algorithm. Notice that this context is not strictly necessary but helps developers to understand what has changed and where. The extent of the context might also be relevant. Novice programmers, for instance, will ask for a much bigger context as they do not have the ability to recognize code parts using only few lines and conciseness is only an hindrance to them. Expert programmers, as well as the authors of the original code, would probably prefer to focus only on the pieces that have changed.

S2: Programmer browsing code history: in other cases a programmer could be interested only in seeing which parts of a file have been changed, from a higher perspective. That is relevant for instance when discussing global refactoring of source code, or when tracing changes of different users, or looking for unchanged components in a large code base.

The automatic diffing of files is also widely exploited by administrators of systems and networks, whose needs are different:

S3: Sysadmin diffing files: minimizing the size of the delta (between different versions of files) is an important requirement for sysadmins, since they expect to store a lot of differences and do not want to waste space. On the other hand, the readability of the output is equally important, since differences are expected to be read primarily by human users. Sysadmins are probably more interested in reading changes line-per-line, with a layout similar to the ones they are used to deal with.

S4: Sysadmin transmitting diff information: sysadmins may also have strong constraints on the available bandwidth and space. Consider, for instance, a group of distributed sensors that collect data (i.e. monitoring temperature, pressure, levels of water and so on) and transmit them to a centralized server. The frequency of updates is extremely high and produces a huge stream of data. On the other hand, it does not make sense to re-transmit unchanged values, neither to add contextual and not relevant information. It is preferable to express differences in the most compact way, especially if they are meant to be processed only by software agents. Such optimisation issues are particularly relevant for multi-sited version control systems, in which the revisions are not controlled by a centralised server but are rather distributed across multiple servers. These have to coordinate each other and to propagate a large amount of data in order to ensure consistency and integrity[18][19].

There are many other situations where detected changes are mainly meant to be interpreted by human readers. In that case, the most effective algorithms are those that can detect specific types of changes:

S5: Author revising literary documents: Consider the case of an XML document where a bold style is added to a fragment by wrapping it into a new element. That change could be recognized as the deletion of the text fragment and the insertion of the new element that contains a new text node. A more compact and precise delta would instead detect that these two operations are logically interconnected. Verbosity here leads to redundancy, rather than to a more accurate representation of the change. Similarly, the split of a paragraph (when a return is inserted dividing a single paragraph into two ones) could be precisely detected as a single operation or as two independent changes, i.e. the deletion of the whole second paragraph, and the insertion of some words in the first one. The detection of higher-level changes would produce more useful delta for users revising literary documents, where similar operations happen very frequently.

S6: Developer working on visualization tools: The previous considerations also apply to developers who build tools working on deltas. For example, the developers of visualization tools must analyze the content of a delta to create the corresponding graphical representation; having deltas that contain domain-specific changes may allow the generation of more precise and interesting visualizations. Additionally, compact deltas can be a problem because they could be difficult to interpret and to make sense of.

Finally, some other requirements exist for diff algorithms running on structured data and knowledge bases:

S7: XML database admin comparing records: an admin that compares different versions of the same records in an XML database deals with specific operations. In that case, in fact, the edit operations often consist of deletions and insertions of entire subtrees or of modifications of text nodes and leaves. In a few cases the database schema changes and the content is wrapped, split or moved around. More than detecting such higher-level changes, it is then important to minimize the size of the delta and to tame the computational complexity. Contextual information is also less relevant here than in other cases.

S8: XML designers comparing attributes: similar considerations can also be extended to all those scenarios where changes are expected to affect textual content, more than structures. Consider, for instance, an XML vocabulary where most information is stored in attributes

(one example is GXL [20]). The modification of an attribute value can be expressed in different ways: detecting exactly which characters changed, detecting the deletion/insertion of the textual value of the attribute, detecting the deletion/insertion of the whole attribute or even detecting the deletion/insertion of the whole element containing that attribute (with its value changed). The first solution is the most accurate but the others are faster to calculate. Moreover such accuracy may not be required. For instance, if an attribute contains a date users might be interested in the modification of the overall date, instead of the modification of a single digit of the day, month or year.

From this discussion, it is clear that there is not a single definition of what makes a diff algorithm *good*. The suitability of a diff algorithm depends on the deltas it creates and how fit these deltas are for use in a certain scenario. Deltas that are good in a scenario for certain reasons may be bad in another for the same reasons. There is, thus, the need to create metrics to measure each of the qualities that can be found in a delta.

4. Metrics for quality evaluation

The goal of this work is to define some metrics that capture peculiarities of deltas. These parameters can be used, for instance, to select the most appropriate algorithm for the scenarios discussed in the previous section. Here we introduce the main ideas behind our framework and we identify which information is needed to measure these metrics. In the next sections we will give a formal description of our model and of each metric.

These metrics are meant to capture some qualities that are objective in their nature. It is outside the scope of the metrics to capture other features, such as the ability to reverse a delta, its correctness or its size in bytes in a certain encoding: some of these features are taken for granted (as it is the case with the correctness of a generated delta) or are implementation-specific but not indicative nor fundamental to the general behaviour of a diff algorithm.

As we said, there is an ongoing request to find more appropriate metrics to evaluate the quality of deltas. For example, in [16] the authors stress on the human-interpretability of the deltas and the distinction between high-level and low-level changes. They also measure the quality of the deltas in terms of “intuitiveness” and “conciseness”. We use similar terms for some of our metrics; in the case of “conciseness”, even the same exact term but with a different meaning. In fact, the motivations and goals of these works are very similar.

On the other hand, some fundamental differences are worth highlighting. First of all, the scope: while they focus on diffing ontologies, our metrics are general and can be applied to other domains and data structures (streams of text, trees, lists, etc.); second, they propose a well-defined, unambiguous and complete set of changes on RDF(S) knowledge-bases, while our model is independent on the set of changes the algorithms are able to detect.

Last but not least, the overall structure of the metrics is different. Our metrics aim at getting a deeper characterisation of the changes contained in a delta. Besides separating basic (low-level) and composite (high-level) changes, we try to capture information about the nature of composite changes and composition rules; for instance, we introduce a metric — called *terseness* — to also take into account the amount of contextual information in each change.

Before going any further, let us briefly describe what a delta is in the UniDM model and which information it may contain, as we will build our metrics on top of it. A summary of the formal definitions will be given in Section 5.

Listing 1: Source document (S)

```
<book>
  <info>
    <author>John Doe</author>
    <rights licence="cc-by-sa" />
  </info>
</book>
```

Listing 2: Target document (T)

```
<book>
  <info>
    <author><name>John</name><surname>Doe</surname></author>
    <rights licence="cc-by-sa" />
  </info>
</book>
```

Figure 1: Example documents.

To simplify the explanation of our metrics we will base our examples on two XML documents shown in Figure 1 and some possible deltas generated by algorithms with different characteristics, shown in Figure 2. Please note that to make the examples more readable, many details are not shown in the presented deltas. Most notably, the XML elements are referred to by their names rather than by more appropriate pointing techniques such as XPath [21] or XPointer [22]. The deltas in the dataset accompanying this paper contain all the needed details.

In UniDM a Delta is a collection of changes that an algorithm detects, together with some relations between these changes.

There are two kinds of changes: *atomic* and *complex*. Complex changes are obtained by aggregating atomic ones into more meaningful structures. An example is given in Figure 2: the change C.1 of Delta 2 can be seen as the aggregation of the changes B.1, B.2 and B.3 of Delta 1bis. The detection of moves is a further very common example of complex change. Moving a paragraph from a document, for instance, can be expressed as a single high-level change or as a combination of the deletion of the original paragraph and the combination of another one that is an exact clone but in a different location.

There are many kinds of relations between changes that can be stored in a delta. The two most common kinds are the *application order* relation, used to define a partial or total order in which changes should be applied, and the *grouping* relations, used to bundle different changes that are somehow related. These relations are used to express various properties of a delta, such as its reversibility [23] or the fact that it contains parts that can be independently applied [24].

Thus, we make a distinction between the top level changes in a delta and the other ones, that have been grouped in other complex changes.

Let us now discuss our metrics and their possible applications.

4.1. Length

The dimension of a delta has been widely used to evaluate the quality of diff algorithms, as discussed in Section 2. Two main approaches have been adopted: space consumption [11][6]

Listing 3: Delta 1	Listing 4: Delta 1bis
A.1: REMOVE-TEXT(John Doe)	B.1: REMOVE-TEXT(John)
A.2: ADD-ELEM(<name>,<author>)	B.2: ADD-ELEM(<name>,<author>)
A.3: ADD-TEXT(John,<name>)	B.3: ADD-TEXT(John,<name>)
A.4: ADD-ELEM(<surname>,<author>)	B.4: REMOVE-TEXT(Doe)
A.5: ADD-TEXT(Doe,<surname>)	B.5: ADD-ELEM(<surname>,<author>)
	B.6: ADD-TEXT(Doe,<surname>)
	B.7: REMOVE-CHAR(' ')
Listing 5: Delta 2	Listing 6: Delta 3
C.1: WRAP(John , <name>)	D.1: IDENTIFY-NAME(John , <author>)
C.2: WRAP(Doe , <surname>)	D.2: IDENTIFY-SURNAME(Doe , <author>)

Figure 2: Possible deltas for <author>. For illustrative purposes, elements are referred to using their names rather than a proper pointing mechanism such as XPointer.

and edit distance [4][12]. There are various definitions of edit distance; the most basic one is defined as the number of changes needed to transform one document into another. There are also weighted variants of the edit distance where costs are associated to the kind of changes produced (e.g. ADD has cost 1 while DEL has cost 1.3, making removals slightly more expensive) or to the amount of data modified by the changes (e.g. making ADD("ab") more expensive than ADD("a") but cheaper than ADD("a") + ADD("b")). Note also that the edit distance is highly affected by the set of available operations and their weights.

To avoid overloading the word “edit distance” we refer to the most basic edit distance as Length. The Length measures the number of changes listed in a delta. The goal of many algorithms is to produce deltas with the smallest possible Length, i.e. as short as possible. That is required for instance in scenarios S3, S4 and S7 of the previous section. However, there are cases in which a moderately long delta can be preferred over a shorter, equivalent delta. For example, the editor in S5 may prefer to see a long list of words that have been modified in a paragraph rather than a single change that states that the whole paragraph has been changed. At the same time, she/he does not want an even longer delta in which a change is generated for every single changed character.

4.2. Terseness

There are other evaluations besides the absolute measurement of the space used by the delta. It is interesting, for instance, to understand whether or not that space is actually required or the delta contains redundant information.

That is why we introduce the concept of *Terseness*. The Terseness indicates the relation between the elements actually modified and the elements that appear in the delta, even if they were not modified. The measurement of terseness would be useful in several scenarios. In scenarios S3 and S4 extreme terseness is required. In these cases every repeated byte is a wasted byte. On the other hand, deltas that are too terse are very hard to read and interpret by a human. For this reason some algorithms include a bit of context around the real modification, so that the user can understand better what is being changed and where. In fact, scenarios S1 and S2 would benefit from a lower degree of terseness.

The tradeoff represented by the Terseness metric can be seen clearly comparing two possible ways to express the modifications to the AUTHOR element in Figure 2. Delta 1 removes the subtree rooted on author and adds the new version of the same; delta 2 wraps some characters with elements. The latter is more terse than the former because it avoids using the same data in more than one change or repeating data that is already present in the source document.

4.3. Conciseness

Measuring only length and terseness still hides some useful information about the internal behavior of the algorithms. It is in fact interesting to also discover which strategies an algorithm uses to detect a given number of edits and, in particular, its capability of aggregating atomic changes into complex ones.

Consider a diff algorithm that works on source code and recognizes deletions and insertions of entire lines of code. The deletion of the lines 7-9 could be expressed as a sequence of atomic deletions `DEL-LINE(7)`, `DEL-LINE(8)`, `DEL-LINE(9)` or as a single change `DEL-LINES(7..9)` that groups together the smaller changes. The first representation is more verbose but captures exactly what has happened on each unit of content, while the second one is more compact and provides a high-level view of changes.

For this reason, we introduce the idea of *Conciseness*. Conciseness indicates to what extent the atomic changes found in a delta have been grouped into complex changes. More precisely, it indicates the relation between the number of changes in the top level of a delta (that have not been grouped in other changes) and the overall number of changes. Note that top level changes can be atomic or complex.

A high degree of conciseness implies a reduction of the details exposed in the delta. This reduction usually leads to a clearer representation of changes for humans, where details are hidden behind higher level changes. Consider again the example in Figure 2. The `WRAP` operation (change C.1) in Delta 2 is easier to understand and it is actually derived by aggregating small changes into a bigger one (changes B.1, B.2 and B.3).

That is why a high conciseness is preferable, for instance, in scenarios S1, S2 and S5 where changes are expected to be interpreted mainly by humans. On the other hand, conciseness may be irrelevant or even problematic for software agents. For example, tools that process and apply deltas need to support all these higher level operations. For them, it would be easier to work with atomic changes, though in a very long sequence.

Notice also that the value of the conciseness metric is strongly related to the length metric, but not equal to it. A high degree of conciseness, in fact, implies a smaller length of the delta but it does not necessarily imply that the size of the delta is reduced. For example, large space is required when the delta also carries information about both the aggregating and the aggregated changes. Finally note that two deltas with the same length can have two different degrees of conciseness, depending of the different sets of changes grouped together. Consider the Delta 2 in the example, that could be a refinement of Delta 1 (Listing 3) or of Delta 1bis (Listing 4). If derived from Delta 1, the conciseness is lower since the same delta has been generated from a smaller set of changes.

4.4. Compositeness

There may be other interesting information worth capturing about the behavior of diff algorithms on complex and atomic changes. For instance, it is relevant to measure if an algorithm tends to prefer complex changes more than atomic ones. Conciseness is not enough for this, as

it only evaluates the number of top level changes against the overall number of changes in the delta (without considering top level atomic changes).

That is why we need a new metric, that we call *Compositeness*. Compositeness indicates how much of the delta length is due to the use of complex changes. More precisely, it indicates the relation between the number of complex changes and the overall number of changes (complex or atomic) in the top level of the delta. Note that the compositeness only considers the top level of the delta, without going into details of nested changes.

A high degree of compositeness indicates that a delta is particularly suitable for being interpreted by humans. That is very useful, for instance, in scenarios S5 and S6. The human analysis of literary documents and their evolution, as well as the implementation of tools for capturing and describing changes, are more effective when working on more abstract representation of changes. Similar considerations can also be applied to scenario S2: a programmer interested in a global view of code would benefit a short list of more meaningful changes instead of a longer list of small atomic ones. In case of scenario S1, on the other hand, a programmer might be interested in a more specific and localized view of changes and would also benefit a lower degree of compositeness.

The examples in Figure 2 show two very different levels of compositeness: Delta 1 is composed of atomic changes only, so its compositeness value is extremely low. On the contrary, in Delta 2 all the atomic changes have been grouped into a complex WRAP that lead to a very high degree of compositeness. The same applies to the previous example on deleting lines of code: distinguishing single line deletions produces a very low value of compositeness, while the compositeness of a delta that detects the deletion of a set of contiguous lines is very high.

4.5. Deep Compositeness

The previous metric, compositeness, indicates how much of the delta length is due to the use of complex changes. The compositeness of a delta does not take into account the depth of these complex changes. However, the presence in a delta of deep changes, i.e. changes that encapsulate other changes that in turn encapsulate other changes and so on, is a strong indicator of the fact that the algorithm has been able to associate more meaningful operations to the detected changes.

For example, consider again the case in which some lines of a document have been deleted. We have already said that the grouping of similar changes into a complex change (DEL-LINES(7..9)) increases the compositeness. Let us assume that the lines under investigation form a paragraph. If the algorithm knows the document format and sees that, for example, there are empty lines around the removed lines, it can encapsulate the complex change DEL-LINES(7..9) into a more meaningful change DEL-PARAGRAPH-STARTING-AT(7). The latter change carries more meaning than the former, yet the value of the compositeness metric is the same.

To capture this other perspective on how changes are further encapsulated by an algorithm, we introduce a new metric that we call *Deep Compositeness*.

The expected degrees of deep compositeness to evaluate our scenarios are similar to those of compositeness: higher values for scenarios like S1, S2, S5 and lower ones for situations like S3 and S4. This metric becomes even more relevant when deltas are expected to be read by humans since it measures in a deeper way the ability of capturing abstract changes. For this reason, for instance, a high level of deep compositeness is required for scenario S6, when changes have to be visualized and analyzed separately.

Such a metric is also useful to measure to what extent an algorithm is specialized for specific formats. For instance, an algorithm that knows HTML operations will be able to detect more

complex changes on XHTML files, than an algorithm limited to XML operations or, even, to pure text. Thus, using this metric can be helpful to select an algorithm for scenarios S7 and S8.

5. Formalization of deltas

In order to provide calculable formulas for the metrics previously outlined, we need to establish first a reference delta model. We will use this model to calculate various objective properties of the deltas; the metrics formulas will be built upon these properties.

The delta model we will use in this paper is the Universal Delta Model (UniDM) [9][10]. UniDM is able to express deltas produced by different algorithms on different kinds of documents and based on arbitrary sets of recognized operations. We adopted this model over other existing models [25][3][26][27] because all the other models are limited to a specific document format or force a limited view of what deltas can represent, while the metrics we present in this paper are general and suitable for use with any delta or diff algorithm. The discussion of the completeness of UniDM is out of the scope of this paper. An exhaustive analysis can be found in the original works.

In this section we summarise the main features of UniDM. We will introduce and use only a subset of the whole UniDM model: just enough to provide the formal groundwork on top of what our metrics will be built.

The core of the UniDM is the description of deltas, how they aggregate changes, how changes can be used to expose details about the detection of sophisticated, domain-specific modifications and how basic and format-specific operations can be defined. In addition to this, the model provides also a simple formalization of what documents are and what they are composed of.

5.1. Model of document

To illustrate the point of this paper we will use a simplified version of the UniDM definition of *document*. Documents are seen as graphs where the nodes are the components of the document (e.g. paragraphs and chapters in a DocBook document, nodes and attributes in a XML document, etc.) and the labelled edges are the relations between said nodes (e.g. order relations, containment relations, references, etc.).

Definition 1 (Document). A *document* (D) is a set of elements (E) and relations between elements (R).

$$D \equiv (E, R)$$

This reduced definition of document is flexible enough to describe all kinds of documents at a certain abstraction level (e.g. at the XML Infoset level or at the Unicode string level).

For simple string-like documents only one kind of elements and one kind of relations (follows (e_1, e_2), i.e. e_1 is followed by e_2) are needed, as shown in figure 3. More elaborate documents like XML trees can be represented using different types of elements and relations, as shown in figure 4.

The string “hello world” can be represented as:

$$D_h = (\begin{array}{l} \{e_1, e_2, \dots, e_{11}\}, \\ \{\text{follows}(e_1, e_2), \\ \text{follows}(e_2, e_3), \\ \dots, \\ \text{follows}(e_{10}, e_{11})\} \end{array})$$

where

$e_1 = \text{elem}(\text{h})$, $e_2 = \text{elem}(\text{e})$ and so on, and the function $\text{elem}(v)$ returns a unique element representing the value of v .

Figure 3: The string document "hello world" in UniDM notation.

The XML document `<p class="notice">ready
</p>` can be represented as:

$$D_h = (\begin{array}{l} \{e_1, e_2, \dots, e_5\}, \\ \{r_1, r_2, \dots, r_5\} \end{array})$$

where

and

- $e_1 = \text{xml-elem}(\text{p})$
- $e_2 = \text{xml-attr}(\text{class})$
- $e_3 = \text{xml-text}(\text{notice})$
- $e_4 = \text{xml-text}(\text{ready})$
- $e_5 = \text{xml-elem}(\text{br})$
- $r_1 = \text{attr-of}(e_1, e_2)$
- $r_2 = \text{attr-has-text}(e_2, e_3)$
- $r_3 = \text{child-of}(e_1, e_4)$
- $r_4 = \text{child-of}(e_1, e_5)$
- $r_5 = \text{follows}(e_4, e_5)$

Figure 4: The XML document `<p class="notice">ready
</p>` in UniDM notation.

5.2. Model of changes and deltas

The differences found by a diff algorithm are expressed as a set of changes. Each of these changes describes an operation that must be done on the source document to reconcile one of the found difference, in other words to make the source document more similar to the target document.

Definition 2 (Change). A *change* is a record of the fact that part of the source document S must be changed using operation op with data d in order to produce a patched document S' .

$$c = (op, d) \mid S' = \varphi(S, c)$$

Changes by themselves do not carry enough information to rebuild the target document from the source document. There are various other pieces of information about the changes that must

be recorded for the delta to be useful. The most basic additional information that is needed is the order in which the changes must be applied, or the lack of such an order (i.e., when changes do not depend on each other). In general terms, change relations are objects used to record that a certain relation exists between certain changes. The meaning and the intended effects of a change relation are described by the type of that relation.

Definition 3 (Change relation). A *change relation* is a tuple describing the fact that there exists a relation of type K between the set of changes C_1 and the set of changes C_2 .

$$r = (K, C_1, C_2)$$

The two most common types of change relations are: *application order* and *encapsulation*. Application order relations describe the (partial or total) order in which changes should be applied, for example “the changes c_3, c_7, c_{89} must all be applied before the changes c_4, c_{12}, c_{72} ”. Encapsulation relations record the fact that a change has been detected as the consequence of the detection of other smaller changes, for example “the change c_{34} encapsulates the changes c_{12}, c_{13}, c_{21} ”.

The encapsulation relation provides the basis for one main distinction that can be found between changes: the distinction between *atomic* and *complex* changes. Some of the generated changes are considered *atomic changes* because they are found and generated by the algorithm looking only at the content of the source and target documents. *Complex changes*, instead, have been generated by analyzing some of the changes that have already found. The link between the generated change and the changes used to generate it are recorded through *encapsulation relation*.

Definition 4 (Encapsulation relation). An *encapsulation relation* is a kind of change relation that links a container change to all changes it was generated from.

$$R_{encapsulation} \equiv \{r \mid r \in R, r.K = encapsulates\}$$

Definition 5 (Atomic change). An *atomic change* is a change that does not encapsulate any other change.

$$C_{atomic} \equiv \dot{C} \equiv \{c \mid c, y \in C \wedge \nexists r : r \in R_{encapsulation}, \\ r = (encapsulates, c, y)\}$$

Definition 6 (Complex change). A *complex change* is a change that encapsulates at least one other change.

$$C_{complex} \equiv \bar{C} \equiv \{c \mid c \in C, \forall r : r \in R_{encapsulation}, \\ r = (k, c_1, c_2), c_1 = \{c\}, c_2 \neq \emptyset\}$$

Note that complex changes might encapsulate either complex or atomic changes.

Definition 7 (Top level change). A *top level change* is a change that has not been encapsulated in any other change.

$$C_{TopLevel} \equiv \hat{C} \equiv \{c \mid c \in C, \forall r : r \in R_{encapsulation}, \\ r = (k, c_1, c_2), c \notin c_2\}$$

Figure 5 shows the overall structure of UniDM in UML and highlights the hierarchical classification of changes, their aggregation in deltas and their relation to other referenced changes. Please note that this is just one of the many possible UML materializations of UniDM. In this particular graph we give a special status to certain encapsulation relations, creating particular classes (CompleChange, GroupingChange, etc.) for changes that encapsulate other changes in certain specific ways. Other relations like apply-before have been left as generic associations, as they are not important for the analysis done in this paper.

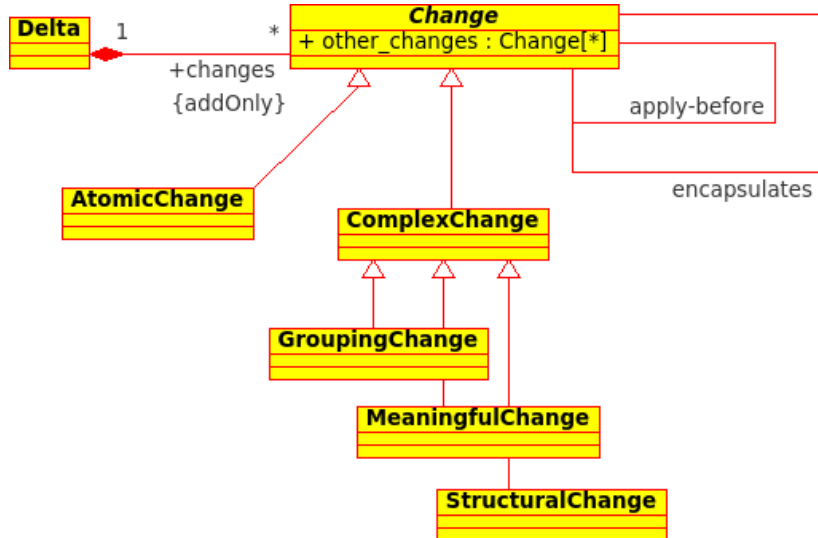


Figure 5: Class diagram of changes in UniDM

UniDM specializes complex changes in three different classes:

Grouping changes Changes that aggregate instances of the same change operating on contiguous elements of the document. An example of a grouping change is `DEL-LINES(7..9)`, that groups together smaller changes to single lines (`[DEL-LINE(7), DEL-LINE(8), DEL-LINE(9)]`).

Meaningful changes Changes that attach a more precise meaning to the aggregated changes. For instance, the removal of a piece of text in a certain position (`DEL("bye")`) and the addition of exactly the same text in the another position (`ADD("bye")`) can be expressed as a more complex change `MovE("bye", 4, 9)`.

Structural changes Changes that aggregate all changes applied to the same document structure, and organize them in a way that resemble that structure or the way users modified it. For

instance, the removal of a XHTML node $\text{DEL}(\langle \text{span class}=\text{"emph"} \rangle \text{text} \langle / \text{span} \rangle)$ aggregates the removal of all pieces of the subtree rooted in that node: $[\text{DEL}(\text{"text"}), \text{DEL}(\langle \text{span class}=\text{"emph"} \rangle \langle / \rangle)]$.

Finally, a delta is a collection of changes and change relations:

Definition 8 (Delta). A *delta* $\delta_{S,T}$ is a tuple of changes (C) and change relations (R) that describes how to transform the source document (S) into the target document (T).

$$\delta_{S,T} \equiv (C, R)$$

Deltas are used to group together the changes found by a diff algorithm during or after the comparison of two documents. As such, they may be regarded as the main output of a diff algorithm but also as the working object used by the algorithm during its computations.

Figure 6 shows the delta 2 of figure 2 codified as a UniDM delta.

$$\text{delta2} = (\{c_1, c_2, \dots, c_8\}, \{r_1, r_2, \dots, r_6\})$$

$c_1 = (\text{remove-xml-text}, \text{"John"}, \langle \text{author} \rangle)$	$r_1 = (\text{apply-before}, \{c_1\}, \{c_2\})$
$c_2 = (\text{add-xml-elem}, \langle \text{name} \rangle, \langle \text{author} \rangle)$	$r_2 = (\text{apply-before}, \{c_2\}, \{c_3\})$
$c_3 = (\text{add-xml-text}, \text{"John"}, \langle \text{name} \rangle)$	$r_3 = (\text{apply-before}, \{c_4\}, \{c_5\})$
$c_4 = (\text{remove-xml-text}, \text{"Doe"}, \langle \text{author} \rangle)$	$r_4 = (\text{apply-before}, \{c_5\}, \{c_6\})$
$c_5 = (\text{add-xml-elem}, \langle \text{surname} \rangle, \langle \text{author} \rangle)$	$r_5 = (\text{encapsulates}, \{c_7\}, \{c_1, c_2, c_3\})$
$c_6 = (\text{add-xml-text}, \text{"Doe"}, \langle \text{surname} \rangle)$	$r_6 = (\text{encapsulates}, \{c_8\}, \{c_4, c_5, c_6\})$
$c_7 = (\text{wrap}, \text{"John"}, \langle \text{name} \rangle, \langle \text{author} \rangle)$	
$c_8 = (\text{wrap}, \text{"Doe"}, \langle \text{surname} \rangle, \langle \text{author} \rangle)$	

Figure 6: Codification of delta 2 from figure 2

5.3. Objective properties of changes and deltas

There are various objective features that can be extracted from a delta and the changes it contains. The most basic one is the number of changes produced by an algorithm; for algorithms that are able to recognize complex changes, one can also extract the number of such changes and rate their complexity. Structural information about the hierarchical organization of changes can also be measured automatically. The main properties that can be extracted from each change expressed in a UniDM delta are:

population the total number of changes of which a change is composed of, including itself and the recursive closure of the encapsulated changes;

depth the length of the longest path from the change to an atomic change, in the graph of its encapsulated changes;

width the number of distinct changes encapsulated directly inside the change;

num touched elements the number of distinct elements of the input documents that are *mentioned* in the change or in its encapsulated changes; in fact, each change might include not only the elements it has actually modified but also some other elements, which provide contextual information (for instance, the lines before/after a modified line in a line-by-line source diff);

num modified elements the minimum number of elements that must be modified by the change to fulfill its purpose; in other words, the number of elements included in the change since they were actually modified; the set of modified elements is then a subset of the touched elements, which does not include those elements added just to provide contextual information;

Note that the nature of touched and modified elements – and the granularity of the overall count – depends on the concrete application domain: it might be single character, a line, an XML element, and so on;

There are also features that can be calculated on the whole delta, with a direct count or by combining measures of individual changes:

num top-level the number of changes that are not encapsulated in any other change;

population the sum of the population property of all changes;

num touched elements the sum of the touched elements of all changes;

num modified elements the minimum number of distinct pieces of information that must be modified in order to turn the original document into the modified one.

From these basic properties, other specialized properties can be derived taking into account only certain kinds of changes:

prop_k is the property **prop** calculated taking into account only changes of kind **k**. For example **num-top-level_{complex}** is the number of *complex* changes that are not encapsulated in any other change.

Table 1 reports the values of these properties on the four sample deltas shown in Figure 2. Most of them have been calculated by aggregating the values of individual changes composing the delta.

	population	# touched	# modified	# top-level (complex)
<i>Delta 1</i>	5	17	3	5 (0)
<i>Delta 1bis</i>	7	17	3	7 (0)
<i>Delta 2</i>	5	11	3	2 (2)
<i>Delta 3</i>	7	11	3	2 (2)

Table 1: Measuring delta indicators on the sample deltas.

To clarify this, let us discuss how the values in the first two rows have been obtained:

Delta 1: since the delta includes only five atomic changes and none of them is encapsulated in any other, its population is 5 as well as the number of top-level changes. The number of complex

changes (fourth column, in parenthesis) is 0 since all changes are atomic. The value of touched elements takes into account the number of characters and XML elements contained in the delta. Since the delta lists 15 characters (the string “John Doe” in change A.1, “John” in A.3 and “Doe” in A.5) and 2 elements (“name” and “surname”, in changes A.2 and A.4) the total value is 17. The value of modified elements indicates the minimum amount of items to change in order to obtain the same output of the delta. It is equal to 3 corresponding to: 1 character (the empty space to delete between “John” and “Doe”) and the new XML elements “name” and “surname”.

Delta 1 bis: the values of the population, top-level and complex changes are respectively 7, 7 and 0 since all changes (B.1-B.7) are atomic and top-level. The value of touched elements is 17 as in the previous case but derives from different touched items: 15 characters (the string “John” repeated in changes B.1 and B.3, “Doe” repeated in B.4 and B.6, the empty space “ ” in B.7) and 2 elements (“name” and “surname”, in changes B.2 and B.5). The number of modified elements is calculated as in the previous case.

6. Formalization of metrics

Now that we have a reference formalization of deltas and their properties, we can give a formal description of the metrics we outlined in Section 4 and calculate them through mathematical formulas.

6.1. Length

The *Length* of a delta indicates the number of changes listed in that delta. Note that changes which have been previously aggregated in complex ones are not relevant for the measurement of the length. In fact, it only considers top level changes.

$$Length(\delta) = \text{num-top-level}(\delta)$$

The values of *Length* for the deltas in Figure 2 are: 5 for delta 1, 7 for delta 1bis, 2 for delta 2 and 2 for delta 3, corresponding to the values in the last column of Table 1.

6.2. Terseness

The *Terseness* of a delta measures the ratio between the number of elements that have been included or referred to in the delta because they have changed and the number of context elements that appear in the delta but were not actually modified. It basically measures the amount of contextual information.

$$Terseness(\delta) = \frac{\text{num-modified-elements}(\delta)}{\text{num-touched-elements}(\delta)}$$

Considering the modified and touched elements reported in Table 1, the values of *Terseness* are: 0.17 ($\frac{3}{17}$) for delta 1 and delta 1bis, 0.27 ($\frac{3}{11}$) for delta 2 and delta 3.

6.3. Conciseness

The *Conciseness* of a delta indicates how much the delta has been simplified by the encapsulation of changes. It indicates the ratio between the number of changes in the top level of a delta and the overall number of changes. The formula has been designed so to return values ranging from 0, for deltas that are not concise, to 1, for very concise deltas.

$$\text{Conciseness}(\delta) = 1 - \frac{\text{num-top-level}(\delta)}{\text{population}(\delta)}$$

The values of Conciseness are: $0 (1 - \frac{5}{5})$ for delta 1, $0 (1 - \frac{7}{7})$ for delta 1bis, $0.6 (1 - \frac{2}{5})$ for delta 2 and $0.71 (1 - \frac{2}{7})$ for delta 3.

6.4. Compositeness

The *Compositeness* of a delta shows how much of its conciseness is due to the use of complex changes. It indicates the ratio between the number of complex changes and the overall number of changes in the top level of the delta.

$$\text{Compositeness}(\delta) = \frac{\text{num-top-level}_{\text{complex}}(\delta)}{\text{num-top-level}(\delta)}$$

The values of Compositeness are: $0 (\frac{0}{5})$ for delta 1, $0 (\frac{0}{7})$ for delta 1bis, $1 (\frac{2}{2})$ for delta 2 and $1 (\frac{2}{2})$ for delta 3.

6.5. Deep Compositeness

The *Deep Compositeness* of a delta measures how an algorithm encapsulates complex changes into other complex changes. The deep compositeness values ranges from 0, for deltas that contain no complex changes and thus no deep compositeness, up to 1, for deltas with many deep changes. The formula has been designed to asymptotically approach the maximum value, so to accommodate deltas with arbitrary average depth \bar{d} .

$$\text{DeepCompositeness}(\delta) = 1 - \frac{1}{1 + \log(\bar{d})}$$

where

$$\bar{d} = \frac{\sum_{c \in \hat{C}} \text{depth}(c)}{\text{num-top-level}}$$

The values of Deep compositeness are: $0 (1 - \frac{1}{1+\log(1)})$ for delta 1 and delta 1bis, $0.52 (1 - \frac{1}{1+\log(3)})$ for delta 2 and $0.58 (1 - \frac{1}{1+\log(4)})$ for delta 3.

7. Applying the metrics

The main goal of these metrics is to allow the evaluation of the fitness of an algorithm in a certain context, by analyzing the characteristics of the deltas it produces. In this section we explain how to apply these metrics to existing algorithms and we present the results of their application on three well-known XML diff tools.

7.1. A two-phase process to evaluate algorithms through metrics

The delta model and metrics are independent from a specific data format. This makes it possible to compare different deltas, produced by different algorithms. The basic idea is to rely on UniDM as reference model to translate all deltas, as depicted in Figure 7. The process can be split in two phases:

1. *interpretation*: the output of each algorithm is mapped into UniDM;
2. *evaluation*: atomic indicators and aggregated metrics are measured on the pre-processed deltas.

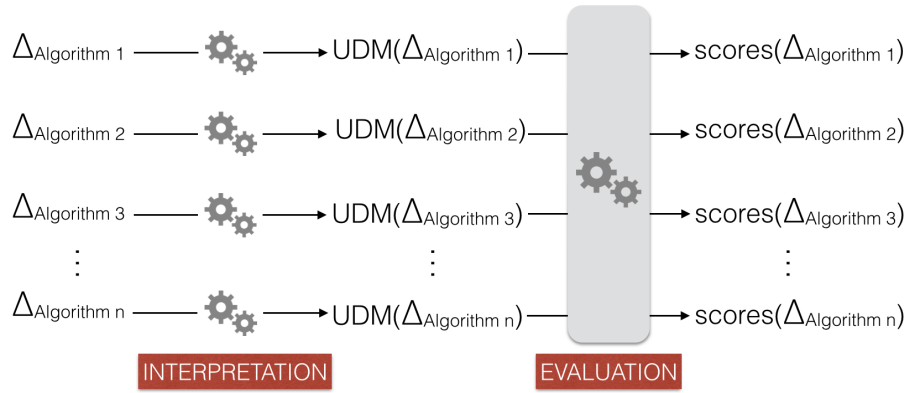


Figure 7: Overview of the process of applying metrics to diff algorithms.

While the *evaluation* step is generic and the same calculation can be applied to all deltas, the *interpretation* one is different for each algorithm.

This is a typical problem of conversions based on an universal model: it is necessary to implement separate converters, each able to manage the specific features of each algorithm and to extract information mapped into the general model; the translation has to be lossless, with no impact on the original algorithm.

The difficulty is then is to build such algorithm-specific converters. In the next section we will briefly explain how we have implemented some converters for XML diff algorithms. The problem, in fact, cannot be discussed from a general point of view but it is strongly tied up with the peculiarities of each algorithm under investigation.

7.2. Applying the metrics to XML diff algorithms

We implemented our approach on three well-known XML diff tools: JNDiff [7], XyDiff [6] and Faxma [2]. To complete our evaluation we also included an implementation of the trivial algorithm for diffing XML files that, when diffing documents A and B, produces a delta with two operations: the deletion of the whole document A and the insertion of the whole document B. Our goal is to verify if the metrics highlight the ‘bad’ behavior of such algorithm: even if correct, in fact, the output of this algorithm provides too little information and is very difficult to use.

The overall evaluation process is an instance of the general two-phase approach described in the previous section, as depicted in Figure 8.

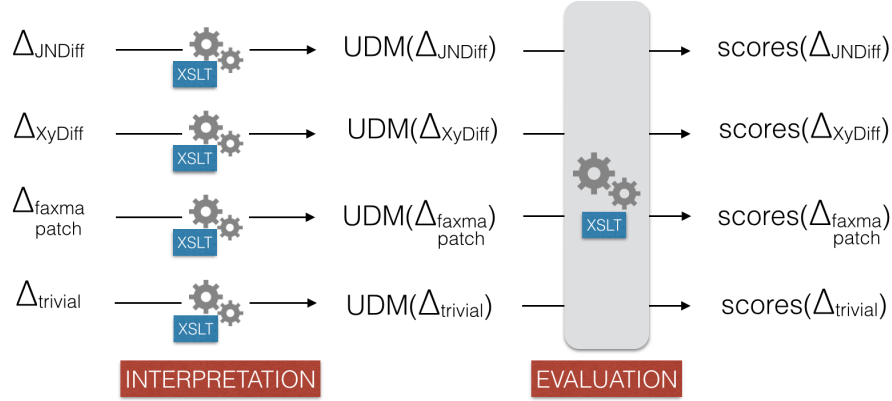


Figure 8: The chain of tools for applying the metrics to XML diff algorithms.

Since all algorithms produce deltas in XML, we implemented converters as XSLT transformations¹. The UniDM deltas have been serialised in a XML format, whose syntactic details are not relevant for our discussion.

The critical point was that the deltas produced by each algorithm are *optimized* for the application that is expected to process them and contain data that will be eventually used to re-build the newer document from the older one. That means, for instance, that changes are ordered in a way that does not necessarily match the order they were applied, rather the order expected by the application; or that some types of changes are aggregated at the end of the delta (for instance, all those involving attributes in the case of most XML diffs).

We studied the serialisation format of both JNDiff and XyDiff and implemented ad-hoc UniDM converters. Basically, the changes found in the original deltas, almost all atomic changes, have been grouped in complex changes similar to those used internally by these algorithms.

The case of Faxma was more complex. The algorithm, in fact does not generate a sequence of edit operations but a format that uses XPath-like expressions to refer to the unchanged fragments of the original document and, among them, interposes the newly inserted elements and attributes. This guarantees a very limited use of memory and resources (that is what authors wanted) but makes it impossible to identify the deleted content from the patch. In this case, the information needed to apply the metrics is totally absent, not only hidden. Furthermore, in the Faxma's deltas there is no way to understand the difference between an update and an addition of content, concepts that are present in the algorithm and in the implementation but are not made explicit in the serialized file. The only solution was to patch the original code of Faxma and to add such missing information. The patched code produces a low-level dump of the changes detected internally. It is worth stressing that this version does not affect the internal functioning of the original algorithm but it only adds some information to the output, logging data that had been removed for optimisation purposes. Both the original and the patched code are open-source². This lead us to discuss a further issue: the possibility of instrumenting the source code (to trace internal delta information) is not applicable to non-open source tools; this is a limitation of our

¹Experimental data and UniDM converters are available at <http://diff.cs.unibo.it/delta-metrics/>

²The original code is available at <https://github.com/ept/fuego-diff>; the new code is available at <https://github.com/gioele/fuego-diff>

approach which can only be applied to algorithms whose code is available or whose serialisation format is expressive enough to be translated in UniDM. This is the case, however, of many existing diff algorithms, as discussed in the previous works about UniDM[9][10].

Finally, note that implementing the UniDM converter for the trivial algorithm was very simple, as well as the tool for calculating metrics; the latter basically extracts data by using XPath and performs simple math calculations, following the formulas presented in the previous sections.

7.3. Experimental results on XML diff

Our experiment consisted of executing all four XML diff algorithms on a set of documents and calculating the metrics on their output, by using the above-mentioned tools. We used the same test-set used to evaluate JNDiff³. It consists of five documents, each available in two versions. These documents are very heterogeneous for size, internal structure and source:

- The first two documents are taken from the evaluation of DocTreeDiff [3]: the first one (identified as LETTER from now on) is a one-page letter while the second is a bibliography of about 15 pages (identified as BIBLIO in the rest of the section). They are both in the XML format used by Open Office 2.x and available on the Web.
- Two others, respectively called DL1184 and DL2221, are XML-encoded legislative acts and bills. They are highly structured in articles, clauses and paragraphs and follow precise rules to encode textual content.
- The last one, identified as PROTOCOL, is an XHTML document containing the specification of a web protocol, used for a classwork project. It is structured in sections and subsections and contains a lot of internal references and code snippets.

Table 2 shows in detail all results we collected. Each column represents a metric. Rows are clustered in five groups, one for each input document. For each document, the table shows the value of each metric calculated on each algorithm. To better highlight differences between algorithms and some common behaviors we summarized these results in the radar charts that follow in this section.

In order to draw these radar charts and to compare the algorithms directly we performed some normalisation on the values of length.

Given that the length value indicates the total number of top-level changes each delta is composed of, in fact, absolute values of deltas produced by different algorithms vary a lot and cannot be compared directly. To solve the problem, we first calculated the magnitude of each length, as $L_{algorithm} = \log_{10}(Length(\delta))$ and then calculated:

$$LengthNormalized(\delta) = 1 - \frac{L_{algorithm}}{\max(L_{JNDiff}, L_{XyDiff}, L_{faxma}, L_{trivial})}$$

Thus, we obtained a *length normalized* score between 0 and 1, with higher values for lower (normalized) values of length. This normalization is also the reason why, for each document,

³The test-suite is available at <http://diff.cs.unibo.it/jndiff/tests/>

		Length (Normalized)	Terseness	Conciseness	Compositeness	Deep Compositeness
BIBLIO	<i>JNDiff</i>	0.45	0.23	0.39	0.13	0.05
	<i>XyDiff</i>	0	0.09	0.01	0	0
	<i>Faxma</i>	0.99	0.09	0.95	0.21	0.08
	<i>Trivial</i>	0.89	0.07	0	0	0
LETTER	<i>JNDiff</i>	0.20	0.38	0	0	0
	<i>XyDiff</i>	0	0.28	0	0	0
	<i>Faxma</i>	0.2	0.44	0.58	0.15	0.06
	<i>Trivial</i>	0.81	0.18	0	0	0
DL1184	<i>JNDiff</i>	0.23	0.27	0.59	0.23	0.08
	<i>XyDiff</i>	0.70	0.05	0	0	0
	<i>Faxma</i>	0	0.06	0.69	0.18	0.07
	<i>Trivial</i>	0.85	0.03	0	0	0
DL2221	<i>JNDiff</i>	0.26	0.43	0.64	0.38	0.12
	<i>XyDiff</i>	0.15	0.08	0.28	0.21	0.08
	<i>Faxma</i>	0	0.07	0.79	0.26	0.09
	<i>Trivial</i>	0.83	0.03	0	0	0
PROTOCOL	<i>JNDiff</i>	0.21	0.26	0.52	0.16	0.06
	<i>XyDiff</i>	0	0.06	0.25	0.07	0.03
	<i>Faxma</i>	0	0.06	0.56	0.09	0.04
	<i>Trivial</i>	0.89	0.06	0	0	0

Table 2: Evaluating metrics for all algorithms on all files of the dataset

there is always an algorithm with length normalized value equal to zero (the one with highest length).

Note that the normalization process is applied *a posteriori* and does not impact the actual measurement of the length. It is only used to compare algorithms, not to calculate the metrics for each of them. It is true, on the other hand, that such a normalisation has to be repeated whenever a new algorithm is considered (as we need to calculate again the denominator of the previous formulas). We think this is acceptable since the calculation is simple and the metrics are calculated just once for each algorithm. We could have instead maintained the original value or could have used a fixed normalisation value (for instance, the length of the delta produced by the trivial algorithm). That would be easier to calculate but would make the radar charts less clear and meaningful.

A clarification is also needed about the terseness values. The terseness indicates the ratio between the modified elements and the touched ones. We approximated the number of modified elements as equal to the minimum amount of modified textual content between the two input documents. This approximation allowed us to calculate the value in a fully automatic and repeatable way, without requiring us to manually inspect deltas. This value in fact can be calculated easily by using external binary diff tools on the isolated characters in the XML text nodes; this value is not totally accurate but it provides a reasonable round-up, since any other algorithm cannot have a higher terseness (as it cannot modify a lower number of characters).

The results on documents BIBLIO, PROTOCOL and DL2221 are graphically shown in Figure 9. These documents are written in three different formats and differ a lot in terms of internal structures and dimensions. Nonetheless all the web graphs for a certain algorithm look similar,

while there are big differences in the graphs generated by different algorithms. This is an important finding: the algorithms show a quite regular behavior and the metrics are able to capture that behavior correctly.

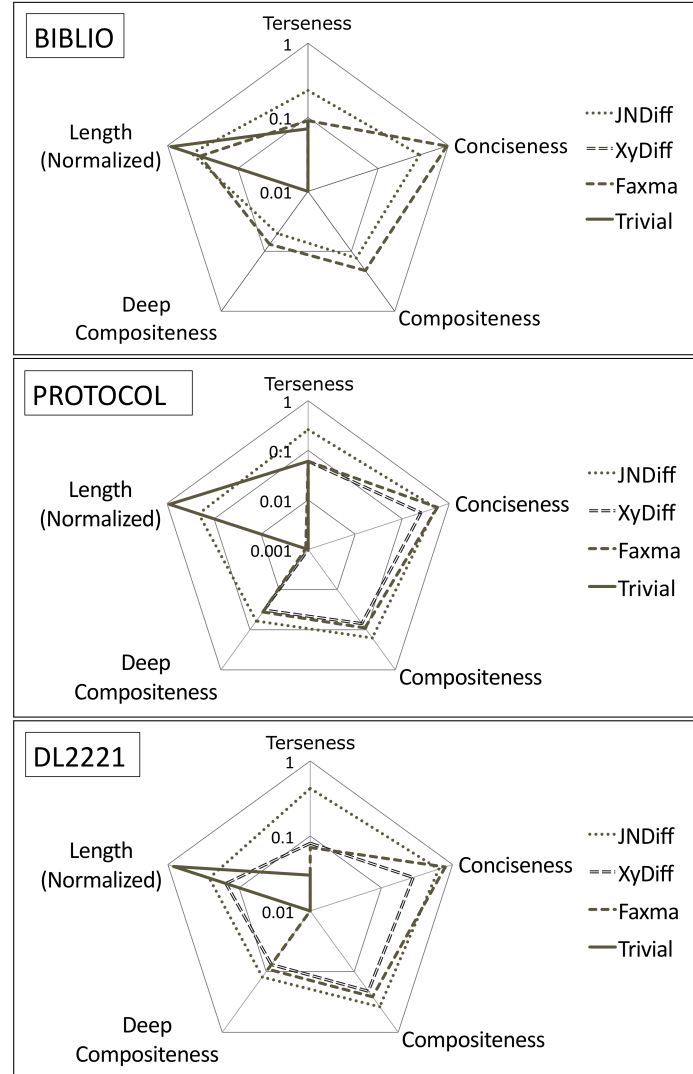


Figure 9: Evaluating metrics on documents BIBLIO, PROTOCOL and DL2221

The other interesting point is that these plots highlight clearly some peculiarities of each algorithm. First of all, consider the results of the trivial algorithm. It scored a very high length normalized (since only two changes were detected) but obtained a zero score for conciseness, compositeness and deep compositeness since it does not try to give a higher-level interpretation of changes. For the same reason, its terseness score is the lowest of all algorithms. Note that a mere evaluation of the number of edits, i.e. of the length metric, would have given a very high

score to the delta produced by this algorithm.

Consider also the behavior of Faxma with regard to conciseness, compositeness and deep compositeness. These dimensions are related to each other and capture whether or not an algorithm is able to aggregate changes into complex ones. Conciseness is very high for Faxma since the algorithm builds large complex changes in the form of “move” changes. These “move” changes are the mechanism used by Faxma to move big chunks of documents without the need to delete and re-add the same data in different position. Notice also that this does not mean that the majority of changes are very meaningful: rather, the complex changes generated by Faxma are just large containers of smaller similar changes; Faxma also tries hard in generating as few changes as possible. These two factors lead Faxma to produce deltas with low values for compositeness and for deep compositeness, though these values are higher than other algorithms that aggregate less changes into more complex ones. However, this helps Faxma with two of its aims: first, it can store fewer changes in the patch because moves are translated as “copy” operations in their patch format; second, the lack of an additional step where changes are interpreted at higher levels allows for more speed and less required space. The normalized scores of the Length metric highlights how the ability of Faxma to generate less changes using copy operations, however, depends on the file structure and the kinds of modifications made to the file: it generated very few changes for an highly structured document such as BIBLIO, but had to generate a lot of small changes to describe the modifications made to semi-structured documents such as LETTER or PROTOCOL.

The low values of XyDiff in almost all metrics also confirm some of its characteristics. The algorithm, in fact, gives much importance to performance and does only consider the size of the delta as indicator of quality. Being a greedy algorithm, it is not able to refine the already generated changes: for example the deletion of three sibling nodes would appear as three separate atomic change deleting one node, not as a complex change enclosing all the three siblings; this reduces compositeness, deep compositeness and conciseness. Its greedy nature also generates unneeded deletion changes when subtrees are deleted: one change is generated for each hierarchy level of that subtree; this makes the delta redundant and the overall length normalized score very low. Another emergent behavior of XyDiff is that fine-grained updates are often expressed as couples of insertions/deletions of large subtrees with a lot of common parts; this reduces drastically the terseness score.

The results on JNDiff are also insightful. The algorithm, in fact, works very well on textual changes and is able to aggregate fine-grained modifications on text nodes into complex changes. On the other hand, such JNDiff is not equally precise on elements and some structural changes that could be aggregated are left disjoint. This is the reason why results are generally good but there is no clear dominance in any dimension. It is also interesting to note that JNDiff tries to limit the amount of nodes involved in each change, in order to be as much faithful as possible to what authors actually did on the document. This is confirmed by the value of terseness, that is the highest in all cases. Similarly, the fact that JNDiff tries to reduce the number of detected edits influences the length normalized score that is always quite high, but lower than the trivial algorithm, as expected. Last, the not-so-high deep compositeness metric shows that JNDiff is indeed able to associate some meaning to the found changes but it still needs some improvements to handle deeper nestings and domain-specific changes.

The experiments on the other two documents produced slightly different results. A deeper analysis, however, shows that even these results are consistent with what we discovered so far about metrics and algorithms’ peculiarities. The plot related to DL1184 is shown in Figure 10.

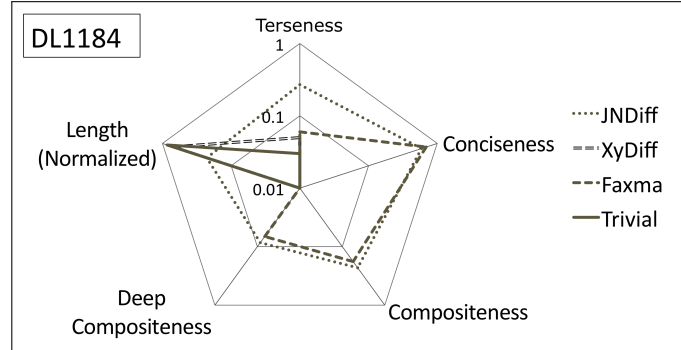


Figure 10: Evaluating metrics on document DL1184

The plot appears different from the previous ones: the length normalized value of XyDiff, in fact, is much higher than the others. Though, we expected an opposite behavior from an algorithm that tends to repeat information and to not express abstract and concise changes. These results depend on the nature of changes applied to the document: a lot of small structural changes on a single flat element. While all other algorithms tend to fragment that change into smaller ones (obtaining a higher number of edits), XyDiff detects a large change without being too precise in detecting sub-changes. This implies that the length value is very high while all other metrics are low. In this case, the internal strategies of the algorithm fits very well with this test case because it has a record-like structure that is similar to that of a database, the class of documents XyDiff has been designed for.

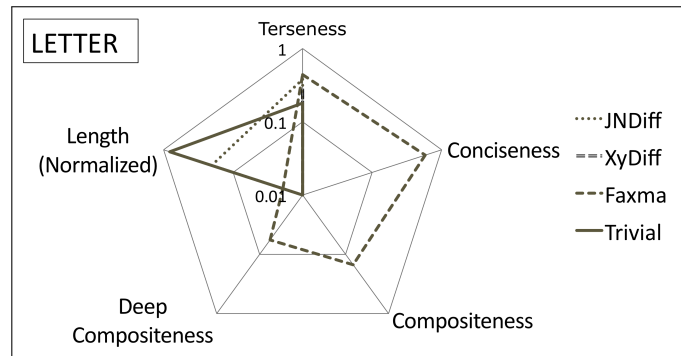


Figure 11: Evaluating metrics on document LETTER. Note that XyDiff is not plotted properly because all the values are near 0.

The final plot, related to document LETTER and shown in Figure 11, looks again very different from the others. The reason is that the changes applied to this document are of specific types, mainly attributes updates, moves and a few changes on text elements (and no one on mixed content-models). The metrics work quite well on these specific types of changes. In fact, the terseness of Faxma is very high. The algorithm is designed to natively detect moves and aggregations of sequences after the deletion of interposing elements. On the other hand, the results of JNDiff get worse: its capability of detecting changes on mixed content-models is not very helpful in this context. JNDiff is not able to aggregate atomic changes into more complex ones.

That is the reason why conciseness, compositeness and deep compositeness are equal to zero.

For similar reasons, since the impact of nested changes and fine-grained textual modifications is very low, XyDiff has the same behavior for these metrics but its values of terseness is a bit higher than in the previous cases.

These experiments showed us some very interesting trends. In fact we managed to find some peculiarities of the algorithms directly from the metrics values, although these values were calculated only from the deltas and without any knowledge of the internals of each algorithm. Though the evaluation dataset was quite small, the direction looks very promising and we expect to obtain a full characterization with a larger analysis.

By limiting us to the data gathered during this evaluation, however, we can propose a ranking of XML diff algorithms to use in the scenarios presented in Section 3. Though not directly applicable to scenarios S1 and S2, since users in those scenarios usually compare non-XML source code, the values of low terseness and high conciseness would suggest Faxma as first choice for similar cases, followed by JNDiff. For these scenarios XyDiff would perform very badly. JNDiff is also the best choice for scenarios S3 and S4, since it guarantees a very good terseness and length (normalized), while Faxma is preferable as second choice. Faxma and JNDiff are also suitable for scenarios S5 and S6. They, in fact, guarantee a good degree of compositeness and deep compositeness. We would prefer to use JNDiff because of its higher degree of terseness (that still maintains enough contextual information for human reviewers), though the higher degree of conciseness of Faxma makes it a good choice too. For the same reasons, JNDiff and Faxma are well ranked for scenarios S7 and S8 as well. Note also that XyDiff is not suggested for scenarios S7, although it should, because we did not, on purpose, take performance into account in our metrics. Finally, note that the trivial algorithm performed bad in all metrics (apart from the length normalized score) thus it cannot be suggested for any of the scenarios under discussion.

The findings of this experiment can be generalized into a set of rules for selecting the most appropriate algorithm for a given situation. Table 3 shows the guidelines we distilled. It describes some common situations and contains one row for each of them. The right column lists the desired values of our metrics for each situation, so that any algorithm showing those values is a good candidate for that case.

When diffing text-based documents, for instance, it is useful to have a clear output for humans, in which high-level changes have been detected from low-level ones (high conciseness and compositeness). An algorithm with a high degree of terseness is fine for experienced users, since they do not need a lot of contextual information. A low degree of terseness (i.e. more contextual information) is instead needed for average users. The ability of an algorithm to reduce as much as possible the information stored in the delta is useful when designers need to reduce space consumption and when diffing large amount of data. When performance is a key aspect, in fact, it is recommended to rely on algorithms with a low degree of conciseness and compositeness, which do not use time and resources for interpreting and reorganizing changes.

Scenario/constraints	Desired metrics
Diff on text-based documents for experienced reviewer	High conciseness High (deep) compositeness High terseness
Diff on text-based documents for average user	High conciseness High (deep) compositeness Low terseness
Need to reduce space consumption	High terseness
Large amount of data-centric documents	Low conciseness Low (deep) compositeness High terseness

Table 3: Guidelines to select the most suitable diff algorithm.

8. Conclusions and future works

In this paper we presented a set of metrics and a methodology for the evaluation of the deltas produced by diff algorithms. The metrics are based on a general delta model, called UniDM and summarised in Section 5. The metrics were successfully experimented on XML diff: our experiments showed that the values of these metrics reflect some properties of the analyzed algorithms, for example the capability of detecting many localized small changes instead of fewer large changes.

The current metrics are only some of the possible metrics one can use to measure the quality of a delta. In our opinion this set gives good indications about the deltas and the algorithms, and their ability to work in a given context. It is interesting, on the other hand, to investigate additional metrics and to verify which new metrics can be built on the same formalization.

A key aspect of our solution is that it is designed not to be bounded to any specific domain. We experimented it on XML diff algorithms and this analysis has not required us to do any modification or extension to the model. The generality of UniDM and our first experiments make us optimistic about the possibility of extending and tuning the metrics to evaluate diff algorithms specialized, for instance, on database dumps[13], ontologies[8] or diagrams[28]. This is one of the main future directions of our research. It will also be interesting to study the relation between these domain-independent metrics and the domain-specific delta models and properties.

Another distinctive point of our metrics is that their values can be calculated in a totally automated way, without resorting to any human intervention or judgment. The fact that these values can be calculated in an unsupervised way opens the way to additional exploitations of these metrics. One possible application is the use of these metrics as a fitness function in genetic algorithms to calculate the best parameters for parametrized diff algorithms. For instance, JNDiff has a threshold parameter that indicates the percentage of content that needs to be changed in a block of text to make the algorithm emit a single large update change instead of many smaller changes. Now this parameter must be set manually by the users of JNDiff; with the use of the presented metrics it would be possible for a user to say “find the highest threshold value that makes JNDiff generate deltas with a high conciseness value”.

References

- [1] S. Faisal, M. Sarwar, Temporal and multi-versioned XML documents: A survey, *Information Processing & Management* 50 (1) (2014) 113 – 131. doi:<http://dx.doi.org/10.1016/j.ipm.2013.08.003>.
URL <http://www.sciencedirect.com/science/article/pii/S0306457313000939>
- [2] T. Lindholm, J. Kangasharju, S. Tarkoma, Fast and simple XML tree differencing by sequence alignment, in: D. C. A. Bulterman, D. F. Brailsford (Eds.), *ACM Symposium on Document Engineering*, ACM, 2006, pp. 75–84.
- [3] S. Rönnau, G. Philipp, U. M. Borghoff, Efficient change control of XML documents, in: U. M. Borghoff, B. Chidlovskii (Eds.), *ACM Symposium on Document Engineering*, ACM, 2009, pp. 3–12.
- [4] Y. Wang, D. J. DeWitt, J. yi Cai, X-diff: an effective change detection algorithm for XML documents, in: U. Dayal, K. Ramamritham, T. M. Vijayaraman (Eds.), *Proceedings of the 19th International Conference on Data Engineering*, March 5-8, 2003, Bangalore, India, IEEE Computer Society, 2003, pp. 519–530.
- [5] J. Jacob, A. Sachde, S. Chakravarthy, CX-DIFF: a change detection algorithm for XML content and change visualization for WebVigiL, *Data & Knowledge Engineering* 52 (2) (2005) 209–230.
- [6] G. Cobena, S. Abiteboul, A. Marian, Detecting changes in XML documents, in: R. Agrawal, K. R. Dittrich (Eds.), *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA, USA, February 26 - March 1, 2002, IEEE Computer Society, 2002, pp. 41–52.
- [7] A. Di Iorio, M. Schirinzi, F. Vitali, C. Marchetti, A natural and multi-layered approach to detect changes in tree-based textual documents, in: J. Filipe, J. Cordeiro (Eds.), *Enterprise Information Systems, 11th International Conference, ICEIS 2009*, Milan, Italy, May 6-10, 2009. *Proceedings*, Vol. 24 of *Lecture Notes in Business Information Processing*, Springer, 2009, pp. 90–101.
- [8] D. Zeginis, Y. Tzitzikas, V. Christophides, On computing deltas of RDF/S knowledge bases, *ACM Transactions on the Web* 5 (3) (2011) 14.
- [9] G. Barabucci, Introduction to the universal delta model, in: *Proceedings of the 2013 ACM Symposium on Document Engineering, DocEng '13*, ACM, New York, NY, USA, 2013, pp. 47–56. doi:10.1145/2494266.2494284.
URL <http://doi.acm.org/10.1145/2494266.2494284>
- [10] G. Barabucci, A universal delta model, Ph.D. thesis, University of Bologna (Apr. 2013).
- [11] C. Percival, Naive differences of executable code, <http://www.daemonology.net/bsdif/> (2003).
URL <http://www.daemonology.net/papers/bsdif.pdf>
- [12] E. Leonardi, S. S. Bhowmick, Xandy: A scalable change detection technique for ordered XML documents using relational databases, *Data & Knowledge Engineering* 59 (2) (2006) 476–507.
- [13] S. Sundaram, S. K. Madria, A change detection system for unordered XML data using a relational model, *Data & Knowledge Engineering* 72 (2012) 257–284.
- [14] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom, Change detection in hierarchically structured information, in: H. V. Jagadish, I. S. Mumick (Eds.), *SIGMOD Conference*, ACM Press, 1996, pp. 493–504.
- [15] S. V. Coox, Axiomatization of the evolution of XML database schema, *Programming and Computer Software* 29 (3) (2003) 140–146.
- [16] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, V. Christophides, On detecting high-level changes in RDF/S KBs, in: A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, K. Thirunarayan (Eds.), *International Semantic Web Conference*, Vol. 5823 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 473–488.
- [17] P. Plessers, O. D. Troyer, S. Casteleyn, Understanding ontology evolution: A change detection approach, *J. Web Sem.* 5 (1) (2007) 39–49.
- [18] J. Zhang, I. Ray, Towards Secure Multi-sited Transactional Revision Control Systems, *Comput. Stand. Interfaces* 29 (3) (2007) 365–375. doi:10.1016/j.csi.2006.05.007.
URL <http://dx.doi.org/10.1016/j.csi.2006.05.007>
- [19] I. Ray, J. Zhang, Towards a New Standard for Allowing Concurrency and Ensuring Consistency in Revision Control Systems, *Comput. Stand. Interfaces* 29 (3) (2007) 355–364. doi:10.1016/j.csi.2006.05.008.
URL <http://dx.doi.org/10.1016/j.csi.2006.05.008>
- [20] A. Winter, B. Kullbach, V. Riediger, An overview of the gxl graph exchange language, in: *Revised Lectures on Software Visualization, International Seminar*, Springer-Verlag, London, UK, UK, 2002, pp. 324–336.
URL <http://dl.acm.org/citation.cfm?id=647382.724795>
- [21] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, J. Simeon, XML Path Language (XPath) 2.0, Recommendation, W3C, <http://www.w3.org/TR/2007/REC-xpath20-20070123/> (Jan. 2007).
- [22] P. Grosso, E. Maler, J. Marsh, N. Walsh, XPointer Framework, Recommendation, W3C, <http://www.w3.org/TR/2003/REC-xptr-framework-20030325/> (Mar. 2003).
- [23] D. Shapira, M. Kats, Bidirectional delta files, *Inf. Process. Manage.* 48 (3) (2012) 587–597.
- [24] J. Dagit, Type-correct changes: a safe approach to version control implementation, Master's thesis, Oregon State University (2009).

- [25] J.-Y. Vion-Dury, A generic calculus of XML editing deltas, in: M. R. B. Hardy, F. W. Tompa (Eds.), ACM Symposium on Document Engineering, ACM, 2011, pp. 113–120.
- [26] A. Cicchetti, D. Di Ruscio, A. Pierantonio, A metamodel independent approach to difference representation, *Journal of Object Technology* 6 (9) (2007) 165–185.
- [27] M. Klein, Change management for distributed ontologies, Ph.D. thesis, Vrije Universiteit Amsterdam (Aug. 2004). URL <http://www.cs.vu.nl/~mcaklein/thesis/>
- [28] S. Wenzel, D. Poggenpohl, J. Jürjens, M. Ochoa, Specifying Model Changes with UMLchange to Support Security Verification of Potential Evolution, *Comput. Stand. Interfaces* 36 (4) (2014) 776–791. doi:10.1016/j.csi.2013.12.011. URL <http://dx.doi.org/10.1016/j.csi.2013.12.011>