Alma Mater Studiorum · Università di Bologna

Dottorato di ricerca in
Informatica

Ciclo XXV

Settore concorsuale di afferenza: INF/01
Settore scientifico disciplinare: 01/B1

A universal delta model

Presentata da: Gioele Barabucci

Coordinatore dottorato:
prof. Maurizio Gabbrielli

Relatore:
prof. Fabio Vitali

Esame finale anno 2013

Contents

Chapter 1

Introduction

The only constant is change.

Heraclitus

More and more contemporary human knowledge is stored in electronic documents, see the enormous number of web pages, text documents, photos and music files spread all over the hard discs of the world, or centralized repositories of knowledge such as Wikipedia or OpenStreetMap. All these documents are seldom static, they are constantly changing: sometimes they are changed by their original authors, other times they are collectively edited, yet other times they are modified by automated bots. Regardless of who or what makes these modifications, what can be observed is a continue status of flux of this mass of documents: they are constantly being changed.

The study of how these documents change can provide deep insights about how knowledge and culture evolve. The history of edits made to notable pages on Wikipedia contains a treasure trove of interesting facts: changes in the way a topic is seen in the media, modifications brought by the passing of time, corrections to factual data to reflect new discoveries, etc. The ability to analyze how documents change is a key point to be able to understand how knowledge has changed and how it may change in the future.

A common way to analyze how documents change is the extraction of differences between two documents, usually between two revisions of the same document. The document that describes the differences that have been found is called the *delta* between the first document, the *source*, and the second document, the *target*.

There are fields in which the deltas themselves are important and subject of debate. Deltas are the basis of the code review process employed by many software project, especially open source projects. In projects that use code reviews, modifications to the source files are not directly committed by the contributor to the core repository. Instead, the changes are submitted to the project discussion lists as deltas and reviewed by other members of the project that can approve them or offer advice on how to improve the proposed modification so it can be accepted. In this case, deltas are used to concentrate the attention of the reviewers on what has changed, sparing them from having to review the whole codebase every time a modification is proposed. Other examples of the importance of deltas as standalone documents can be found in philology and textual criticism in particular. Textual criticism studies the differences between different copies of a manuscript to try to reconstruct a version of the manuscript that is a near as possible to original intent of the author. The differences between these copies are analyzed instead of the complete manuscript, putting in evidence the parts of the documents that provide the most interesting clues.

Deltas can be generated in three ways: by writing by hand the list of modifications to be done to a document, by recording the actions of the author of the modifications while they modify the document or by comparing two files containing two versions of a document, the source and the target version. The first case, the direct creation of the deltas can be seen in the field of law making: acts usually do not create new laws but amend the existing ones. For example, the US Copyright Term Extension Act changed the text of the US copyright law extending the duration of the copyright protection from 75 years to 95 years. The fact that the content of the act focuses only on what is being changed and do not repeat the complete text of the law, highlights the message of the lawmakers: "we have not changed the copyright regime, we only changed this specific aspect of it, the length of the protection". The second case, tracking the actions of the author, is the way word processing tools manage the changes introduced during the editing process. The traces of the user's actions are seen by these tools as the description of what has changed, regardless of what has been changed in the content of the document. The third case, the comparison of two revisions of a document, is the normal way to extract a list of things that have been changed if no explicit summary of the modifications exists and there is no access to the traces left by the author while modifying the document. This is the way in which differences between source code files are customarily shown; this is also the only technique that makes it possible to compare documents that are just representations of other documents, for example electronic representations of ancient documents stored on parchments.

Comparing documents to extract a list of changes is not a novel idea, but the ability to perform these tedious comparisons with automatic tools allows the development of interesting systems. Take for example the use of versioning systems, systems that keep track of all the versions of a document, allowing the retrieval of older versions. The feasibility of these systems is in big part due to the fact that they do not need to store complete copies of the tracked files, only the deltas between two subsequent versions. This greatly reduces the amount of data that must be stored or transferred in order to save a new version. Another example is the generation of the stemma codicum of a manuscript, i.e. the study of all the available copies of a manuscript to infer which copy has been copied from which so to trace how culture used to spread in a certain historical period through the analysis of repeated copying mistakes or adjustments. To generated a *stemma codicum*, a philologist has to analyze many different versions of the same document and to figure out what is the best arrangement of these copies in terms of what has been copied from what: are there strong clues that the copy A of the manuscript has been copied from the copy B, that in turn has been copied from the copy C or does it make more sense to think of both A and B as independent copies of the common ancestor C? These questions must be answered for all the copies that are being analyzed and this may mean dozens of possible comparisons. The use of automatic comparison tools relives the scholars from the need to perform these comparisons by hand, allowing them to focus on analyzing the small deltas that have been found instead the complete documents. Yet another example of processes that can be simplified by the use of automatic comparison tools is the creation of amendments during the debate of a bill in a parliament. Before a bill is approved and becomes and act, members of parliament and other authorities can submit amendments to the proposed text. These amendments are instructions on how to change the text that is being debated (e.g., “remove comma n. 4” or “modify ≪lifetime pension≫ to read ≪4-year pension≫”) that are voted by a legislative assembly. If an amendment is approved, the text of the bill is changed as specified in the amendment. The amendments are usually written by hand by the staff of the member of parliament that wants to propose them. The creation of these amendments require careful drafting because they must obey to all the formal rules about their content and their style before they are eligible for discussion in the parliament and can be voted on. An alternative to this cumbersome and error-prone task is the generation of amendments from a modified copy of the original text. First the member of parliament edits the text of the bill, then this modified text is compared with the original text, then a delta is extracted and, finally, an amendment is generated by rewriting the changes found in the delta using the rules prescribed by the legal system. This technique, that is faster than the manual way of creating amendments and ensures that all the legal rules are followed, cannot be implemented without the availability of automated comparison tools.

The core of the automated comparison tools are the *difference algorithms*, also called *diff algorithms*. The purpose of a diff algorithm is to compare two documents, the source and the destination document, and to produce a third document, the so called *patch* that describes in precise terms which changes must be done to the source document to make it become identical to the target document. For example, if the source document is the sentence “the sky is blue” and the target document is “the night sky is dark”, a diff algorithm may produce the patch “add ‘night’ after the first word; replace ‘blue’ with ‘dark”’.

Many different algorithms have been designed over the years. The main motivation behind the creation of new algorithms is, historically, the desire to have algorithms that run faster or use less memory. In the recent times, however, other algorithms have been designed to address different problems. First, new algorithms started taking into account some qualities of the generated patch: for instance the number of change instructions needed to express the found modifications or the readability of the patch itself. In addition to this, various specialized algorithms have been designed, to compare documents stored using the same file format or conceptual model. These specialized algorithms exploit their knowledge of files formats to identify changes more precisely. Specialized algorithms are also able to ignore small modifications that do not changes the meaning of the content for examples the amount of white space in XML documents.

1.1 Research problem

Although all diff algorithms operate in a similar way and produce conceptually similar deltas, it is hard for tool makers to move from an algorithm to another, to enable the use of multiple algorithms in the same tool or to analyze the behavior of a set of algorithms to understand which one fits better a certain situation.

In principle, all the diff algorithms operate in the same way: first, similar portions of the two compared documents are detected and aligned, second, the different parts are analyzed and changes are generated, then the changes are refined in order to optimize the generated delta and, finally, these changes are saved in a certain format, to be used by patching tools or other tools. Also in principle, the data structures used by these algorithms are all the same, or, at least, they are all exported in the same way.

In practice, however, all the existing algorithms operate in their own peculiar way: they use their own vocabulary to express what they are doing and which changes they have detected, they also use different data structures and completely different serialization formats to store the found changes.

This heterogeneity makes it hard to compare how the algorithms work and to isolate the parts that they have in common from those in which they differ. It is also hard for tool-makers to change the used algorithm once a tool has been built based on a particular algorithm.

It is reasonable to say that all the discussed problems can be traced back to a main issue: the lack of a shared formal model of documents and deltas. If algorithms and tools were based on such a shared model, it would become much easier to change existing algorithms (for instance, to support a little variation of a document format), to compare the behavior of these algorithms and also their output, to write tools able to use more than one diff algorithm.

1.2 Research goal

The main goal of this thesis is to answer the following questions:

- Is it possible to formalize what it means to find differences between documents?
- Is it possible to find a single shared formalization that can be used by any diff algorithm working on any kind of document?

1.3 Proposed solution

The proposed solution is a model that abstracts and formalizes all the parts that are common to most diff algorithms:

- The concept of document (What is a document made of? What does it mean that a document is in a certain "format"? What about the fact that some documents carry the same knowledge although they have different content?).
- The concept of delta and changes (What exactly are these change objects detected by diff algorithms? How can we bundle these changes together, so that they form an identifiable unit? Are there changes that express more meaning than others?).
- What changes can be detected (Do all the algorithms recognize the same set of changes? Are certain changes related to other changes? What about changes that make sense only when comparing documents of a certain format?).
- How to achieve interoperability between the algorithms that produce changes and the tools that apply them (How should these changes be stored when returned through an API? How should they be written into files that are meant to be used by different tools? How to deal with the fact that some tools are able to generate changes that may not be understood by other tools?).

This proposed model provides a groundwork on top of which it is possible to create new algorithms, redefine the existing ones and analyze them with automatic tools.

The fact that this model is not only a conceptual model but also a mathematical formalization makes it possible to write tools that perform analysis that previously could only be carried out by humans.

1.4 Structure of the dissertation

This dissertation is structured in three parts: problem, solution and applications.

The first part introduces the problem of the heterogeneity of the delta models used by the algorithms described in literature. Specifically, chapter 2 describes the problem in detail and with practical examples while chapter 3 reviews many models found in literature, not only explicit models but also implicit models emerging from the code or from the output of the existing algorithms.

The second part is dedicated to the model proposed as a solution to the research problem. This second part begins with an overview of the model in chapter 4 and continues with formal and informal definitions of the concepts that compose the model. Chapter 5 illustrates with a simple meta-algorithm the abstract structure that can be found behind any diff algorithm. Chapter 6 describes the concept of "document" and formalizes the relation between all the format that are used, one on top of the other, to encode the knowledge carried in the document itself. Chapter 7 formalizes the concept of delta as a set of changes to be done to a source document to turn it into the desired target document. Chapter 8 formalizes what a "change" is, how changes in the same delta are related to each other and how they can be classified. Chapter 9 presents a list of properties of changes and deltas that provide objective data about the changes that have been found. Chapter 10 describes the concept of "operation", i.e. what is the meaning of a change and how the document is to be modified when a change is applied; later, chapter 11 presents a catalog of possible changes, starting from basic universal operations and concluding with domain-specific operations. Last, chapter 12 shows various data structures for internal use or in APIs that can be used to encode the formalism previously described; these data structures are models as UML classes.

The third part presents some practical applications of the model. Chapter 13 gives an overview of what applications are made possible by the adoption of this models. Chapter 14 describes in detail how to evaluate, in an automated and objective way, the "quality" of the deltas produced by an algorithm, relying on some of the properties made explicit by the model. In the same chapter, it is shown how it is possible to show the behavior of an algorithm through the deltas is produces, without looking at its code. Chapter 15 is, instead, devoted to the presentation of a technique for the identification of the various phases that occur during the development of an ontology (for example the initial creation of the class hierarchy, the later refinement, the documentation of certain properties, etc.). This identification is possible thanks to a novel diff algorithm for OWL ontologies, also presented in the same chapter; the development of this diff algorithm has been made substantially easier by the adoption of the proposed delta model.

The last chapter, chapter 16, summarizes the new scientific findings contained in this dissertation and positions the proposed model with regards to the existing state of art.

The dissertation ends with a bibliography of published results and a sitography of resources available over the web.

Part I

Expressing deltas

Chapter 2

A missing piece: a universal delta model

One of the problems faced by tools that rely on comparison algorithms is that it is hard to swap the used algorithms in favor of another or to integrate two or more algorithms at the same time. The main reason behind this problem is that different algorithms produce deltas in different formats and using different delta models; in practice the main drawback is the big amount of changes required to move from using an algorithm to another. This problem makes it also harder for a tool to support multiple algorithms at the same time, for example to compare the same files at two different abstraction levels. The adoption of a single delta model could make it simpler to change from a comparison algorithm to another; this delta model could be employed directly by algorithms as their own internal model but could also be used as a intermediary model to and from which translate the model used by current algorithms.

There are many reasons why a tool would be interested in changing its underlying algorithm. Historically efficiency has been the main concern behind this kind of changes: in some cases to move to an optimized implementation of the same algorithm already in use (e.g. from MJD Diff [58] to Ned Konz's Algorithm::Diff [62], two implementations of the Hunt-Szymanski algorithm [24]), in other cases to use an algorithm with a different computational complexity or with different trade-offs between comparison speed and required memory (e.g. from [36] to [35]). Another reason for a tool to change the algorithm it uses is the fact that, for some documents, specialized algorithms can produce much "better" deltas or detect more meaningful changes, for example a tool that shows subsequent revisions of a book in the XML-based DocBook format [68] may prefer a tool that can detect specialized changes like "a paragraph has been split in two" instead of "the XML subtree rooted on the `<para>` element has been removed and two subtrees with `<para>` elements as root elements have been added". More recently, license concerns with the library that implements comparison algorithms have become another common reason for replacing the used algorithm.

Algorithms are not only replaced in tools, they are also added along the existing ones so that they can be used together or chosen at runtime by the user. There are two main reasons why the use of a single algorithm in a tool may not be enough. First, users may be interested in different kinds of comparisons: a fast but imprecise comparison, a comparison that takes longer to be performed but reports a smaller and more focused set of changes or, also, a comparison that analyzes only some aspects of a document (e.g. its structure and not its textual content). Tools such as Oxygen XML Diff & Merge [65] provide the user with a choice between these algorithms fulfilling these different requirements, while still showing the found changes in the same graphical interface. Another justification for the need of multiple diff algorithms is that many electronic documents can be seen as expressing different abstraction levels, each requiring its specialized diff algorithm. An example of this are OWL ontologies [59]: physically they are stored as text files, conceptually they express collections of facts about classes, properties and entities but are also meant to be interpreted in more abstract terms as Description Logic axioms, generating interpretation sets and similar mathematical objects [63]. Tools such as Ecco [20] allow users to compare two ontologies at different levels: how do their classes and properties differ? how differently are their entities connected? how do the (possible infinite) generated interpretation sets differ? Each of these levels of abstraction requires a specific diff algorithm. Tools that want to offer a comprehensive representation of the modifications between two ontologies must be able to present a view of the difference for each of the abstraction levels, this means that multiple algorithms must all be implemented in the tool.

Although it is an often-desired operation, it is not easy for toolmakers to replace the algorithms used and the libraries that implement them, mostly because each algorithm uses its own internal conceptual model (i.e. the view of what is a difference and what kinds of differences exists and can be detected) and serialization format (i.e. the data structures used to manipulate the elements of the model). The models and formats used are not based on a commons shared specification. While some formats are more widespread than others, said formats are often limited to their original narrow domain, they are often underspecified and they rely on assumptions buried in the source code of their reference implementation. The "unified patch" format, described at [67], shows all these flaws: its usefulness is limited to line-based textual documents, its model of changes is very basic and is not thoroughly documented, the syntax used to point inside the referenced documents is a byproduct of certain implementations.

The fact that most algorithms use their own model of what is a document or what is a change, creates a conceptual mismatch that must be overcome in order to integrate a new algorithm in a tool. Take, for instance, a tool that shows differences between text files based on the Myers algorithm [35]. The list of changes the tool receives from the algorithm contains changes that tells which lines have been added or removed, where *line* means a string of characters between `\n` characters. Suppose now that this tool wants to use instead bsdiff [64] as its comparison algorithm. First, the bsdiff algorithm works on single characters, not lines. This means that the tool will have to modify the way it interprets the pointer supplied in the changes. Second, the bsdiff algorithm produces ADD (with the meaning of “copy”) and INSERT operations, not the ADD and REMOVE operations that the Myers algorithm produces. This means that the tool will have to modify also the way it processes the received list of changes.

The conceptual mismatch between different algorithms is only one one face of the problem. The other face of the problem is the way the produced information is encoded in data structures sent across the API boundaries, i.e. how the deltas are serialized. Almost no two implementations of diff algorithms share the same API or data structures. Similarly, only few implementations produce output files based on the same file format, the notable exception being line-based text diff tools. As a result of this, tools that use diff algorithms must write different wrappers or code path in order to be able to manipulate the deltas produced by the algorithms, even though these objects are not very different in their content and structure.

An analysis of the existing diff models and format across various domains shows that beneath this sea of incompatible models and formats lies a sizable core of shared notions. For example, most algorithms produce insert and delete operations plus some peculiar operations; the semantics of these operations is similar across different algorithms; the file formats used to save the found deltas they all look very similar. The purpose of my thesis is that it is possible to express in a single model the basic features common to all document formats, fields of application and comparison algorithms. On top of that layers of specialized features can be built, providing a nice way to reduce these higher level features into sets of features describe in the common core.

The idea of creating models for deltas that can be shared between different diff algorithms is not novel: there are several attempts at creating reference standalone models. However, all these models have some shortcomings, analyzed in chapter 3. The main limitation of all these models is that they all focus on a single kind of documents, for example XML (e.g., [47], [41]) or OWL (e.g., [38], [55]).

The solution proposed in this thesis, in chapters 4 to 11, defines a universal delta model based on the definition of a set of concepts related to deltas and the diff process: the definition of what is a document, the concepts of delta and changes, an extensible catalog of possible operations.

Chapter 3

State of the art in document comparison models

The need for a shared conceptual model arises in different contexts and has been already been discussed in various fields. First, a shared model is needed to allow the exchange interoperable deltas between different programs (e.g. diff applications and merge applications). Then, it is useful to decouple the algorithm that compares the documents from its output, making it easy to change the algorithm or the implementation used in a tool. Having deltas expressed in a single model allows also the comparison of the quality and properties of the generated deltas, making it possible to to objective comparisons of the available algorithms from their outputs instead of relying on more subjective analysis. However, regardless of the importance of having a single shared model, most of the current algorithms do not use one such model, relying instead on their own ad-hoc models, often undocumented and embedded in the code of the reference implementation. Shared models have been proposed in literature, but these models are meant for use in a single particular field, for example [41] is limited to deltas on XML documents, [55] focuses only on RDF documents, and [28] only on OWL ontologies.

The existing models found in literature can be divided in two main categories: models embedded in algorithms and standalone models. In order to review the embedded models it is necessary to analyze also the mechanisms behind the algorithm for which they have been defined; the coupling between the two is often so strong that the models cannot be discussed without referring to some details of the algorithms themselves. The review of the standalone models, instead, can be performed more in depth and in more abstract terms with the only caveat of these models being focused on one particular format or one document domain only.

While all these models have they pros and cons, the following analysis provides a solid beginning point for the extraction of a common core of functionalities and the subsequent development of an extensible catalog of operations, all components of the formal model described in the second part of this dissertation, from chapter 4 to chapter 12.

3.1 Models embedded in algorithms

Most of the models used by diff algorithms are not standalone models: some of them are implicit models that arise from the details of an algorithm, other are are small ad-hoc data structures used while processing the documents, yet other are broad descriptions of what can be detected by an algorithm. Regardless of the fact that these models are not documented independently from enclosing algorithms, they are the models used in practice and, as such, it is not possible to propose an alternative to them without first carrying out a thorough analysis of them.

3.1.1 Algorithms for generic documents (sequences)

3.1.1.1 Hunt-McIlroy

The Hunt-McIlroy algorithm [23] is the algorithm used by the original UNIX diff program.

The Hunt-McIlroy algorithm sees documents as sequences of lines. More precisely, documents are seen as sequences of numbers because what the algorithms operate over is an hash of the content of the line, not the content itself.

This algorithm detects three operations: addition, deletion and modification of a line. The detected changes are later serialized in a script for the UNIX line-oriented text editor qed.

The "modifications" changes are not detected by main algorithm, that concerns itself only with solving the LCS problem, but by the serialization phase, where additions and deletions that hit the same non-common contiguous portion of sequence are rewritten into changes.

3.1.1.2 Myers (GNU diff)

The Myers algorithm [35, 33] is the foundation of the GNU diff tool. The Myers algorithm is a greedy algorithm that runs in $\mathcal{O}(ND)$ time and $\mathcal{O}(N)$ space.

Instead of computing, like the previous algorithms, a solution to an LCS problem between the source document and the target document, the Myers algorithm solves a shortest edit script problem (SES), a problem dual to LCS. The length of the edit script is measured by counting the number of changes to be generated, without any additional weighting process. The list of changes to be generated is selected among the many possible by applying a special case of the single-source shortest path to the edit graph. The edit graph, illustrated in figure 3.1, is a graph in which the nodes are all the possible alignment between the elements of the source document and that of the destination document and the edges are the possible transitions between these alignments; diagonal edges exist between identical elements, horizontal edges express that a element is available only in the source document, vertical edges that an element is available only in the destination document.

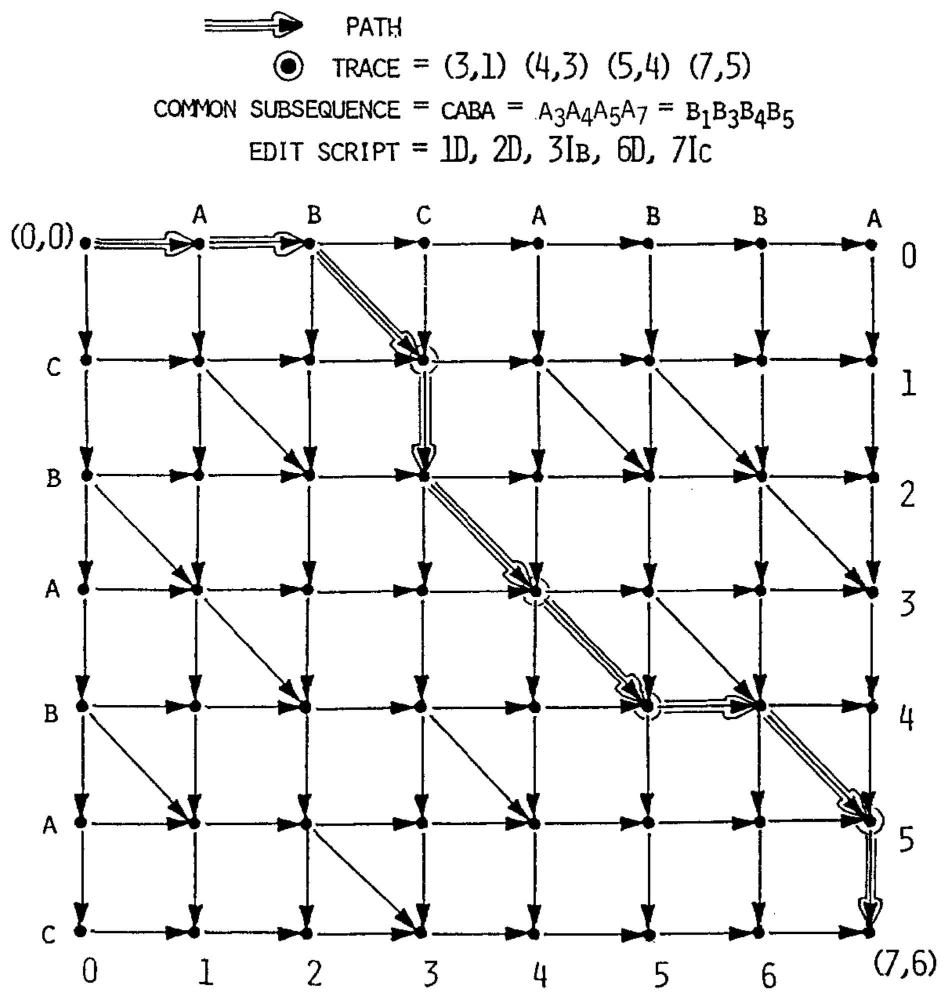
PATH
TRACE = (3,1) (4,3) (5,4) (7,5)
COMMON SUBSEQUENCE = CABA = A3A4A5A7 = B1B3B4B5
EDIT SCRIPT = 1D, 2D, 3IB, 6D, 7IC
(0,0)
A B C A B B A
C B A B A C
0 1 2 3 4 5
0 1 2 3 4 5 6
(7,6)

Figure 3.1: An example of edit graph for Myers algorithm (extracted from [35])

The Myers algorithm can detect two operations: addition and deletion. These operations mirror the moves found by the algorithm over the edit graph: additions are moves over vertical edges, removals are moves over horizontal hedges. During the serialization stage, sequences of consecutive additions or consecutive deletions are grouped together.

3.1.1.3 Burns-Long

The Burns-Long algorithm [11] improves other greedy LCS algorithms such as Myers' [35] by computing in $\mathcal{O}(1)$ a delta that can be serialized to a patch with a near-optimal size.

This algorithm operates on sequences of bytes over which a rolling checksum is calculated. The algorithm uses this rolling checksum to scan the target file and find subsequences that appear also in the source file. All the data in the target file that cannot be matches in the source file is regarded as an addition.

Given the way it operates, the Burns-Long algorithm detects two operations: addition and copy. Instead of emitting deletion changes as most of the other algorithms, The B-L algorithm produces copy changes that move pieces of the a sequence to be kept over a sequence to be deleted, as depicted in figure 3.2. This choice (using overwriting copy changes instead of deletion changes) is fundamental to the speed and memory performance of the algorithm, but has two main drawbacks. First, all the copy operations must be present in the serialized patch, this means that the size of the patch is going to be $\mathcal{O}(N)$ unless some sort of range compression mechanism is used (Burns-Long has one such mechanisms). Second, neither the delta nor the patch can be reversed, i.e. the patches produced by the Burns-Long algorithm cannot be transformed and applied to the target document to obtain the source document. This happens because the copy changes do not contain enough information about the pieces of document that will be overwritten, a piece of information necessary to reverse the delta.

3.1.1.4 bsdiff

The bsdiff algorithm [64] is an $\mathcal{O}((n+m)\log n)$ algorithm that find differences between files at the binary level. Similarly to [11], it produces patches based on two operations: COPY and INSERT.

3.1.2 Algorithms on trees and XML trees

3.1.2.1 Faxma/Fuego diff

The Faxma algorithm [32] is a greedy diff algorithm for XML documents. Like other XML diff algorithms its computational complexity is $\mathcal{O}(n)$ in the best case and $\mathcal{O}\left(n^2\right)$ in the worst case. Faxma tries to balance the speed of comparison, the ability to detect non-basic operations (moves) and the generation of a compact patch.

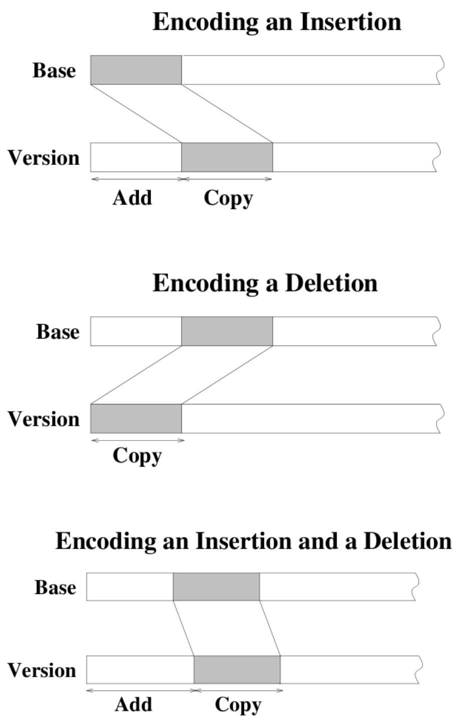

Figure 3.2: Addition and copy operations in Burns-Long (extracted from [11])

Faxma works by aligning sequences of XML XAS event tokens [27] (e.g. StartElement(e), EndElement(e)) instead of performing its computations directly on the trees of the XML documents. Working with XAS events instead of trees of nodes allows Faxma to reduce the problem of matching identical parts of the XML trees to the well-studied problem of aligning sequences. Once the two documents have been transformed into sequences of XAS event tokens, the matching algorithm start refining the list of found changes, looking first for long matching sequences, then trying to refine the list of different sequences by searching for matching sequences of smaller size. The outcome of this process is a list of insertions and copies. However, this generated list is a list of XAS event tokens, not fit to describe what are the found changes in terms of operations on nodes and trees. The found differences are thus later serialized in an ad-hoc format, turning the list of possibly non-well formed elements into a well formed XML document.

The set of operations detected by Faxma as stated in [32], consists of four operations: insertions, deletions, updates and moves. In practice, however, only insertions and copies are produced by Faxma. Delete operations are detected as a by-product of the matching algorithm and are later discarded during the generation of the XML patch as the patch is not meant to be reversible, so is it not necessary to serialize the found deletions. The lack of update operations is due to the fact that the matching phase is only able to find either matching or non-matching sequences of events and has no concept of "same element with different content", as the algorithm only deals with XAS events, not proper elements.

3.1.3 Algorithms for ontologies

3.1.3.1 PROMPTDiff

In [37], Noy et al. present an algorithm for the detection of changes between OKBC ontologies [12], later extended to allow the comparison of RDF and OWL ontologies. PROMPTDiff is a fixed point algorithm that focuses on ontology mapping. The idea is to create a delta that resembles a list of mappings between the structure of the source ontology and the modified structure of the target ontology. This list of mapping is equivalent in practice to a list of operations to be carried out on the source ontology, but highlight the declarative aspect of the comparison made by PROMPTDiff.

PROMPTDiff works directly at the OKBC level (or at the OWL level in later versions), without taking into account the serialization format used to store the ontology on disk.

The mappings produced by PROMPTDiff, i.e., the detected operations, are:

- add/delete, for addition and deletion of axioms;
- split/merge, for single concepts that have been split in multiple concepts and viceversa (e.g., the case of the Wine class being split in three classes White wine, Rosé wine and Red wine);
- map, equivalent to a copy instruction.

The PROMPTDiff algorithm detects these mappings using a set of heuristics, whose application order is stated in a dependency graph to make sure that the rules that produce the most meaningful mappings or that produce mappings used by other heuristics are applied first. As in other similar fixed point algorithms, the computation stop when none of the heuristic rules can be applied to the set of obtained mappings or when there are no more mappings to be further processed.

3.1.3.2 OntoVCS

The OntoVCS library [54] compares OWL ontologies to produce a delta of the changed axioms. The delta produced by OntoVCS are unordered sets of changes. The list of possible changes that OntoVCS can produce is exactly the number of statements defined by the OWL specifications, e.g., ClassDeclaration, DataPropertyDeclaration, AnnotationPropertyDomain, SubClassOf, etc. For each of these statements two operations are defined: the addition of it and the removal of it.

The comparison is carried out at the OWL level, so that it is possible to compare ontologies stored using different serializations. The accompanying patching tool is able to apply the produced deltas to ontologies stored in any format, even a third format different from that of the source and target ontology.

An interesting feature present in OntoVCS is the fact that it records in its patches additional information about the source and target. For example it stores information about the two serialization formats used, the list of imported ontologies (that have no impact on the diff at the OWL level and so could be ignored or discarded), the various IDs used for explicit versioning. This allow the patching tool to produce an ontology that, although not exactly equivalent bit-per-bit with the target document, contains, at least, all the additional information useful for its use with ontology tools.

3.1.4 Algorithms for source code

The problem of finding differences between documents has also been explored in the field of software development and source code files.

The are various algorithms and tools available for finding differences between revisions of source files, specialized in various programming languages: Java [2], Verilog [18], C [34]. There are also programming-language-agnostic algorithms that compare abstract syntaxes trees [22, 30, 26, 40] (the efficacy of which is compared [44]). In addition to source code, algorithms and tools have been proposed for compiled binary files [53] or UML uses cases [19].

It is interesting to note that the model used in these algorithms do no differ much from what is used in the field of XML documents: the basic operations recognized are those that operate on the structure of the graph, later these operations are analyzed in search of clues that can justify the detection of more meaningful changes or the aggregation of a set of changes in a single more structured change. The operations used in the deltas generated by these algorithms have little in common with each other, but this is understandable as each of these algorithms deal with a very precise kind of document, for which there is often an established vocabulary of refactoring operations.

3.2 Standalone models

3.2.1 Rönnau-Borghoff

Rönnau and Borghoff presents in [41] a standalone versioning model for XML document, based on previous similar work [42]. This model defines what an XML document is composed of and a catalog of 3 operations on nodes and sequences of nodes (i.e. addition, delete and update). In addition to these, the model also defines a fingerprinting mechanism to help resolving conflicts during the merging process and an XML patch format. The DocTreeDiff algorithm [43] uses this model for its deltas.

It is manifest that one of the main focuses of this model is to make sure that the delta contains enough information to be used in non-perfect scenarios, for instance when its must be applied to a document that differs from the source document. Indeed, the delta contains enough information that it is possible to reverse it without accessing the original document, as long as the delta does not contain update changes.

This model has three main limitations. First it operates only on XML documents. The second limitation is that its fingerprinting mechanism works on canonicalized XML fragments [56], this means that the deltas produced by the algorithms and tools that used this model cannot be trusted to produce bit-perfect target documents once applied. This is a minor issue because the patched documents are equivalent at the XML level to the target documents. The last, more fundamental issue, is that this model is very focused on elements, making text nodes, attributes, comments and other XML nodes "second-class citizens". For instance, additions of attributes are detected but are reproduced as updates of the containing element; similar things happens with text nodes and processing instructions. The consequence of this focus on elements is that the produced delta is less explicit or understandable than a delta produced under a model that differentiate between the many types of nodes present in an XML document.

3.2.2 The Delta ontology for RDF/N3

The Delta ontology [55] proposes a model the description of deltas for RDF graphs. In this model the differences between two RDF documents are expressed with two sets of operations: insertions and deletions. The two basic operations diff:insertion and diff:deletion are implemented as specializations of a generic diff:replacement operation. This model also defines the difference between weak and strong deltas. Weak deltas are deltas that can only be applied correctly to a document identical (at the RDF level) to the document use as source document to generate the delta; strong deltas, instead, contain enough context information to be applied correctly also to documents that are slightly different from the one used in the original comparison.

The two operations defined by the model (i.e., insertion and deletion) operate only on whole RDF statements, not on single parts of the statements such as the subject or the predicate of a statement. This means that the deltas are a bit coarse, but this also allows for the redefinition of the diff:insertion and diff:deletion operations inside the RDF/N3 framework. In fact, it is possible to formalize the relation between diff:replacement and diff:insertion/diff:deletion using the N3 formula

```
{ ?F diff:replacement ?G }
        log:implies
{ ?F diff:deletion ?F; diff:insertion ?G }
```

Although the set of available instruction is very basic, and so is the way the are defined, the fact that these operations on RDF graphs are fully defined in terms of other RDF + N3 statements allows for potentially interesting manipulations and introspection opportunities.

3.2.3 Klein

In [28] Klein proposes a framework to classify the various changes that can happen during the development of ontologies. The main contribution of [28] is an extensive ontology of changes that describe what changes can happen at the structural level and what other changes can be detected by considering two or more structural changes as part of a more meaningful change. This model is not linked to a particular algorithm, although it has been partially implemented in the PROMPTDiff tool [37].

In the Klein model the output of an algorithm, i.e. the delta, is called the transformation set and is a set of changes that, once applied, turn the source ontology into an ontology equivalent to the target ontology.

The changes that are contained in the transformation set can be either basic or complex changes. The basic changes are additions and removals of basic ontology statements, e.g. class declarations or the specification of the domain of properties. The complex changes are changes created aggregating other basic changes.

3.2.4 Papavassiliou et al.

The papers [50, 38, 51] introduce and develop a framework for the definition of changes in knowledge bases based on RDF/S [60]. The same papers present also an algorithm based on that framework. The underlying delta model divides changes in in two categories: basic changes and composite changes. Basic changes are changes that have a direct impact on the RDF/S graph and a 1-to-1 mapping to modification to the RDF/S semantics that can be inferred from the graph. Composite changes, instead, are changes that have been generated aggregating other basic changes.

The main peculiarity of this model is the way changes are formally defined: a change is a tuple $(\delta_1, \delta_2, \phi)$ where

- δ_1 is the set of the required added triples, i.e. the triples that must be in the target ontology but not in the source ontology,

- $\delta_2$is the set of the required deleted triples, i.e. the triples that must be in the source ontology but not in the target ontology,

- ϕ is the required additional conditions that must be fulfilled before the change can be generated.

An example of composite change that follows this definition is the following Change_Domain operation.

- **Change** $Change\_Property\,(a, b, c)$
- **Intuition** Change the domain of property a
- **Arguments**
  - $b =$ old domain of a
  - $c =$ new domain of a
- δ_1**(Required added triples)** $\forall c \in C : (a, \text{domain}, c)$
- δ_2 **(Required removed triples)**$\forall b \in B : (a, \text{domain}, b)$
- ϕ **(Required conditions)**
  - $(a, \text{type}, \text{property}) \in Cl\,(V1) \wedge$
  - $(a, \text{type}, \text{property}) \in Cl\,(V2) \wedge$
  - $(a, \text{domain}, b) \in V1 \wedge$
  - $(a, \text{domain}, c) \in V2 \wedge$
  - $\neg Cond\,(Specialize\_Domain\,(a, b, c)) \wedge$
  - $\neg Cond\,(Generalize\_Domain\,(a, b, c))$
- Subsumed basic changes
  - $\forall c \in C : Add\_Domain\,(a, c,)$
  - $\forall b \in B : Delete\_Domain\,(a, b)$
- Inverse change
  - $Change\_Domain(a, c, b)$

This particular way of defining the composite changes is the cause of the main drawback this model: given that the definition of the changes contain the rules that must be followed to generate it, it is hard to design an algorithm that is able to detect the same changes but using different detection rules. A way to address this problem would be to separate the definition of the detection rules from the intended meaning of the change itself.

3.2.5 Vion-Dury

The model presented in [46, 47] formalizes a calculus for the application of changes to XML documents. It is composed of various parts: a model of XML documents, a formalization of paths over XML nodes and operations on them, a calculus that describes how to application of single changes to deltas with multiple changes modifies the documents. The changes are based on three basic operations (called "atomic transformations"): INS, DEL, NOP. This model does not explicitly takes into account complex changes for domain-specific purposes, but describes a similar functionality: the composite transformations. Composite transformations are transformation obtained composing smaller transformations applying them one after the other or in parallel.

The field of application of this model is limited to XML documents. This limitation is not only a willful restriction made to focus on a certain class of documents, i.e. XML documents, but it is also of technical nature: the reliance of the model on paths, hierarchical paths, makes it hard to extend this model to other types of documents where the elements are not structured in a hierarchical way or where it is not possible to enumerate and order the elements using a total order, for example in a graph. On the other hand, the use of paths instead of opaque IDs or fingerprints makes it possible to make explicit and rigorous many properties of the patch process. Paths are of key importance, for instance, for the definition of orthogonal deltas (deltas that can be independently applied) or for the transformation of snapshot compositions into equivalent sequence compositions.

An interesting characteristic of the Vion-Dury model is the fact that its formal representation of XML documents mixes the concept of an XML tree with details of its serialization. This mix between the two levels of abstraction can be seen by looking at the order of the attributes. In the XML model the attributes are unordered but one order must be chosen when writing the XML tree to a file. The Vion-Dury model follows a similar approach: it defines a total order for the attributes, preserving the order on which they appear in the file, but ignores this order during the comparisons.

This model lacks an explicit mechanism to define domain-specific operations on top of the available atomic transformations. This is, however a problem that can be easily overcome because the model already describes a mechanism to combine transformations into more complex deltas. The main missing point is the definition of parameters for the complex transformations. Take, for example, a WRAP operation that can be described in terms of additions and deletions of nodes and subtrees. In order to perform its task, the WRAP operation requires the specification of a node to add as the wrapper and the list of paths to be wrapped and the list of the paths of the nodes to be wrapped. The definition of the WRAP operation would also need a way to describe how to map the parameters of the WRAP operations to that of the single INS and DEL operations it is made of. These two features are not available in the model as published in [46, 47].

3.3 Problems with existing delta models

To recap, the flaws found in existing models can be classified broadly according to the following categories:

- focus on a single kind of document,
- are specific to a single algorithm,
- do not address the fact that the same document can be seen at different abstraction levels.

The main limitation of all the existing models is that they are defined only for a particular kind of documents, e.g., line-based text files, XML, OWL ontologies, etc. While this is understandable from the point of view of simplicity and facility of implementation, it also forces the restatement of the very similar same concepts for each new kind of document.

The fact that many models are strictly coupled with a specific algorithm is also a concern. This fact leads to two problems. First, these models are not explicitly defined and documented, thus they must be extracted from the steps of the algorithm or from the code of the implementation. Second, they only model the parts of the delta that are strictly needed by the algorithm in which they are found.

Another problem with most of the models analyzed in this chapter is that they ignore the fact that, in many cases, documents are composed of parts with different "behaviors" and that a document can be seen at different abstraction levels, some of which are compatible and comparable, other which are not. Take for example the case of literary XML documents. They are normal XML documents in which the main content is the text contained in the elements. Diff algorithms for XML documents that want to address the text nodes in a particular way should be able to reflect the fact that the operations on text nodes are different from the operations on content of the text nodes are different from those that can be made on the tree structure formed by the other nodes, mainly element nodes.

The conclusion that arises from the analysis of the state of art is that there is a big overlap of concepts and "view of the world" between all the existing models. These similarities are, however rarely recognized, and, in fact, have not yet been exploited to create a universal delta model.

Part II

The universal delta model

Chapter 4

A formal model of documents, deltas and operations

Keep it simple, make it general,
and make it intelligible.

Douglas McIlroy

As shown in section 3.1, almost no two algorithms share the same underlying model of document or delta yet they are all based around the same principles: documents as sequences of comparable elements, the detection of basic operations, the refinement of these operations based on some additional knowledge of the document format or domain. My main contribution is a single model of document and deltas. This model is able to express deltas produced by different algorithms on different kinds of documents and based on arbitrary sets of recognized operations. It must be underlined that this unified model is not competing with the other existing models; we are not, thus, in the despicable situation illustrated in figure 4.1: what is currently available are not models that try to be universal: either they focus on a single kind of document, or only prescribe their own set of operations to be used or they are just models implicitly defined by the data structures of the diff algorithms. On the contrary, this model offers a way to bridge together all the existing delta models and to make these deltas interoperable.

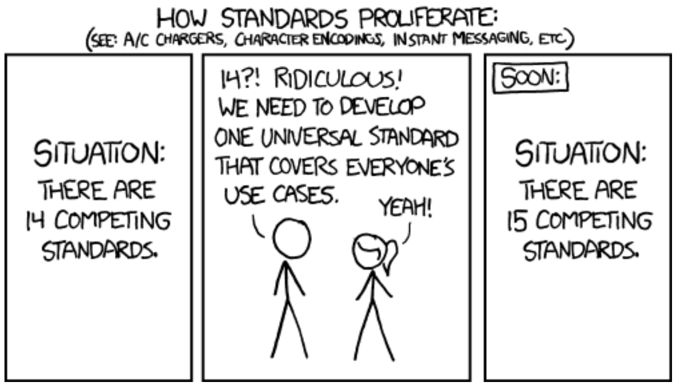
HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Figure 4.1: XKCD comic strip on the proliferation of standards

This model is composed of three main parts:

- a definition of what "a document" is and what makes two documents "the same"
- a definition of what "deltas", "changes" and "operations" are and
- an extensible catalog of operations that diff algorithms could recognize in general or when operating over specific formats or domain.

First of all it is necessary to define what constitutes a document. Without a formal definition of what a "document" is and what is it composed of, it w/ould not be possible for algorithms to refer unambiguously to the parts of it that have changed or have not. It is also important to be able to define precisely when two documents or two pieces of documents are "the same", a non-trivial task when dealing with formats that admit more than one equivalent representation.

The second part of a diff model is the conceptualization of what constitutes a delta, how multiple changes interact with each other and how they are used to turn the source document into the target document.

A formalization of what docs and deltas are would be incomplete and inapplicable without a catalog of possible operations that can be detected between two documents. The purpose of the catalog is to document what are the possible operations and, thus, provide a shared definition and operational semantics of these operations. Aside some basic operations (e.g. addition, removal) and other operations already defined in some domains (e.g. WRAP for trees, CHANGE-DOMAIN for ontologies), it is still an open research question what is the best set of operations to be used. It is not even known what makes a set of operations better than others: some authors strive to find the set of operations that minimizes the produced edit scripts, other focus on how meaningful these operations are, yet others aim at describing operations that fit their domain of application. For this reason the presented catalog is extensible in multiple dimensions and directions: additional format- or domain-specific operations can be added on top of existing operations, new operations can be added to the existing sets, operation can even be removed or ignored while keeping the produced deltas understandable by other tools.

The intended purpose of this universal delta model is to provide a reference model that can be used by any diff algorithm working on any kind of document. This model is meant to be used for both for the formal definition of the algorithms and for their practical implementation, simplifying the study of such algorithms and the adaptation of tools to accept new algorithms.

Chapter 5

Structure of diff algorithms

All the algorithms illustrated in chapter 3 work on the same basic principles or, in other words, there is a single structure that encompasses all these diff algorithm.

```
function DIFF($S, T$)
    $S \leftarrow read\,(source)\,;\quad T \leftarrow read\,(target)$
    $A \leftarrow alignment\,(S, T)$
    $\Delta_{initial} \leftarrow changes\,(S, T, A)$
    $\Delta \leftarrow refine\,(S, T, A, \Delta_{initial})$
    $\Phi \leftarrow serialize\,(\Delta)$
    return $\Phi$
end function
```

Figure 5.1: Generic structure of diff algorithms

This common structure, briefly described in figure 5.1, is composed of the following steps, discussed more in detail in the rest of this chapter.

File input the source and target document are read and interpreted.

Alignment of common parts the parts of the documents that are identical are recognized as such.

Detection and representation of differences the parts of the documents that are not identical are studied and an initial set of changes and change candidates is generated.

Refinement of delta the initial set of changes is refined, choosing the most appropriate changes among all the candidates and aggregating certain changes into more complex changes.

Delta serialization and output the refined set of changes is serialized into a patch file (to be stored or send to other applications) or a number of data structures (to be returned through an API).

The fact that we can identify a single structure for such a variety of algorithms resonates with the idea that it is possible to define a single delta model to be used in any diff algorithm.

5.1 File input phase

The two files that represent the two documents to be compared are read and decoded. The intended purpose of the file input phase is to turn the bits of which the files are composed into data structures that faithfully represent the content stored in those files.

During this file input phase, the content of each file is first read as a sequence of bits, later these bits are interpreted as sequences of information elements. For example block of 8 bits can be interpreted as ASCII characters, strings of variable length (from 8 bits to 32 bits) can be interpreted as UTF-8 encoded UNICODE codepoints, blocks of 128 bits can be interpreted as IEEE 754 [25] quadruple-precision binary floating-point numbers.

Sometimes the decoding phase does not end here: the decoded elements are interpreted again to create another, more structured representation of the content of the document. This is the case, for example, with XML documents. The first decoding step will transform the sequence of bits into a sequence of UNICODE codepoints. These codepoints are then interpreted according to the grammar of XML, generating an XML tree. This last step, generating a new representation based on the representation previously generated of the same file can be repeated many times, depending on the file format used by the files that are being compared and the level of abstraction at which the diff algorithm operates. The stack of the various levels of abstraction of a document and its relation to serialization formats is described in chapter 6.

5.2 Alignment of common parts

Once the two document have been read and turned into appropriate data structures, the diff algorithm starts looking for parts of the documents that are identical or equivalent, so to establish a relation between these common parts, a so called alignment between the elements of the document. The purpose of the alignment phase is to create an initial small working environment for the part of the diff algorithm that detects and processes the changes. After the alignment phase, the algorithm knows which part of the document are of interest: those that have not been aligned (e.g., because they do not appear in one of the documents) or have not been aligned perfectly (e.g., because they have been moved or changed only slightly).

How the alignment phase is performed is one of the two defining characteristics of a diff algorithm, the other being how to classify the found differences.

This alignment can be implemented in a myriad of ways, each with its pros and cons under different aspects. The alignment can be more or less precise (e.g., using heuristics to consider equivalent elements that are 90% similar), require a single pass over the data or many passes (e.g., a greedy algorithm will run in a single pass, but losing the ability to find the optimal alignment), use more or less memory (e.g. to store more the on possible alignment candidate) or, even, tuned to react to certain features of the document content (e.g. preferring the alignment of `<p>` elements over the alignment of `<h2>` elements).

5.3 Detection and representation of differences through changes

After the alignment phase, the diff algorithm knows in which points the two documents differ. It must now encode what it has found into a set of changes, i.e., a set of statements that describe what it is has found in terms of operations to perform on the source document to make it become identical to the target document.

Diff algorithm distinguish one another by the way in which they detect and encode these changes. Take, for example, the two strings “the summer love” and “the lovely summer”. These two strings have various characters in common. A possible alignment is to consider unchanged the strings “The ” and “ ” (This alignment is depicted with underlines in figure 5.2). This is just one of the possible alignments between these two strings, but is one that makes it possible to represent the found differences in many contrasting ways. It is possible to list some of the detectable sets of changes.

1. `The_summer_love`
2. `The_lovely_summer`

Figure 5.2: Alignment of two strings

1. Two word deletions, then two word additions:
    - the word “summer” has been removed,
    - the word “love” has been removed,

   - the word “lovely” has been added in position 4,
   - the word “summer” has been added in position 11.

2. One word deletion, then the addition of two letters, then a word addition:
   - the word “summer_” has been removed,
   - the two letters “ly” have been added in position 8,
   - the word “summer” has been added in position 11.
3. One word motion, then the addition of two letters:
   - the word “_love” has been moved to position 3,
   - the letters “ly” have been added in position 8.
4. One word motion (“summer”), then the addition of two letters:
   - the word “_summer” has been moved to position 9,
   - the letters “ly” have been added in position 8.

These are only few of the possible sets of detectable changes, but enough to illustrate the fact that different algorithms can produce different deltas. In this case, diff algorithms that cannot recognize move operations, for instance because they do not want to spend time looking for possible candidates, will not be able to produce the deltas 3 and 4, although they are shorter and look quite good from the point of view of readability. It must be noted that all these suggested deltas are correct deltas; the difference between these deltas is that some of them may have qualities that can make some user prefer them over others, for example because they are more readable.

If a perfect diff algorithm existed, it would be able to produce a small delta, in a little time and detecting many different kinds of changes. However, this is not possible in many cases because of computational bounds: Zhang et al. [52] demonstrated that in many cases the ability to detect more than two types of changes while producing a minimum number of such changes is akin to solve the *exact cover by three sets problem*, thus NP-hard.

5.4 Refinement of delta

The delta that emerges from the changes detection/representation phase is rarely the delta that is later produced as the final output of the algorithm. Once this first delta is generated, most algorithm refine it through the use of aggregation rules.

For instance algorithms such as Hunt-McIlroy [23] process the initial delta looking for two peculiarities: sequences of similar changes that operate on adjacent lines (e.g., deletion of line 3, deletion of line 4, deletion of line 5) or pairs of additions and deletions that operate on the same line (e.g., removal of existing line 5, addition of a new line 5). When the algorithm discovers that it has generated sequences of similar changes it produces, instead of them, a range change that says, continuing with the existing example, deleted lines from 3 to 5. Similarly when it discovers that two changes operate on the same line, it generates an updated changes instead of a pair of additions and deletion changes.

The rules used to aggregate these initial changes, defined as atomic changes in chapter 8, into more structured changes, defined as complex changes in the same chapter 8, are not always explicit. sometimes they are formally stated and the diff algorithm has a recognizable refinement phase, more often, however, this refinement is done together with the serialization phase and the aggregation rules are hidden in the serialization steps.

5.5 Delta serialization and output

The last phase of any diff algorithm is the serialization of the changes it has found into something that is usable by other applications, either a patch file, to be stored or sent over the net, or a set of data structures, to be used as part of an API.

This serialization step implies the existence of a formally defined set of rules that state how to generate a set of data structures or a sequence of bits. In most implementation of diff algorithms this set of rules has been created ad-hoc, usually mapping the internal data structures to very similar text instruction, for algorithms working on text files, or XML elements, for algorithms dealing with XML files. The only exception to this situation are the algorithms for line-based text documents. The implementations of such algorithms often offer the ability to serialize the found delta using a shared format, the unified diff format [67]. Most implementations of other algorithms however, serialize their deltas using their own serialization format.

5.6 Relation between changes, operations and rules

One of the finest point of this proposed delta model, and of diff algorithms in general, is the relation between the concepts of change, operation and rule. Briefly:

changes are *statements* generated by an algorithm to affirm that it has detected a certain difference;

operations are *names* used in the changes to describe what has been detected;

rules are formalizations of *when* it is acceptable for an algorithm to generate a change that state that a certain operation is to be applied to certain pieces of data.

From the point of view of the diff algorithm, there exists a set of operations that it has been designed to recognize, for example, ADD-WORD, REMOVE-WORD, ADD-LETTERS, MOVE-WORD. From its catalog of operations it knows that the ADD-WORD operation has two parameters w and p, and that the semantics of the ADD-WORD operations is that the word w has been added to the source document in position p, shifting the content present at position $p+1$ to position $p{+}1{+}length(w)$. The algorithms also know the semantics of all the other operations it can detect. In addition to this knowledge about the semantics of the operations, the algorithm also has a set of rules that it applies to the data of the documents being compared and to the set of already found changes to know what are the changes to be generated to describe the differences. For example, the rule for ADD-WORD may be "An ADD(w,p) change can be generated if word w appears document T in position p but not in document S at the same position". Similarly, the rule for MOVE-WORD may be "A MOVE(w,p1,p2) change can be generated if word w appears in document T in position p_2 but not in position p_1 and appears in document S in position $p_1$but not in position p_2". Depending on how the rules are processed and applied, the algorithm will generate, if we consider the example in section 5.3, one of the following deltas:

1. REMOVE-WORD("summer", 4),
   REMOVE-WORD("love", 11),
   ADD-WORD("lovely", 4),
   ADD-WORD("summer", 11)

2. REMOVE-WORD("summer ", 4),
   ADD-LETTERS("ly", 8),
   ADD-WORD("summer", 11)

3. MOVE-WORD(" love", 3),
   ADD-LETTERS("ly", 8)

4. MOVE-WORD(" summer", 9),
   ADD-LETTERS("ly", 8)

All these deltas are correct and the preference for one over the others is a matter of what the designer of the algorithms want to emphasize. For example, an algorithm may be designed to find as many MOVE changes as possible, considering them more concise than pairs of REMOVE/ADD changes, or, on the contrary, to avoid

MOVE changes because they are known to cause problems in tools that will use the generated delta.

From the point of view of a consumer of the delta, the correct understanding and application of the changes is independent from the rules that generated the delta itself. The only requirement for a correct application of the delta is that the diff algorithm that produces the delta and the tool that uses the delta use the same catalog of operations, giving to each operation the same meaning.

Chapter 6

Documents

This chapter defines what a "document" is, what it is composed of and how it can be compared to other documents.

Documents are the mean through which knowledge is stored, transmitted and processed. Although the pure knowledge is the subject of interest to be stored, in order to be processed by automated tools, it must be transformed into a finite, representable set of symbols. The quality of these transformations from pure knowledge to symbols can vary much depending on many factors: experience, familiarity with the domain, suitability of the tools. Knowledge representation [10] and information architecture [49] are two fields of scientific inquiry that deal with the fine aspects of how to go from pure knowledge to more structured symbolic representations of it.

The set of choices one makes about how to structure the pure knowledge in a document forms the so called model of the document. The model describes, for instance, if the information is organized in a linear or in a hierarchical fashion, or what are the possible fields to use to store the information (e.g., the given name of a person goes in the field called "FirstName", not "Name" or "GivenName", while the last name goes in the field called "SName", not "Surname"). The content of the document, shaped according to the chosen model is transformed into a sequence of bits using a format, i.e. a set of rules that states how to map a certain aspect of the model (e.g., a certain field or a value) into a string of bits.

In contemporary documents, format rules do not usually translate models directly into strings of bits. Instead, formats translate from high-level models into lower-level models. A model that is widely used a lower level on top of which other models are defined is XML. Many document formats are now XML-based, this means that the specifications of the document format indicate how to translate their constructs into XML trees, not strings of bits. The translation of these XML trees into bits is delegated to the grammar of XML, not embedded directly in the document format. This situation leads to the vision of a document as a “cake” of stratified abstraction level, each with its own model. This effect can be seen clearly in XHTML documents: at the bottom of this cake lays the bitstream level where all the information is stored as binary digits; on top of that it is possible to see a textual document made of UNICODE characters (the textual level) that, in turn, can also be interpreted as an XML document (the XML level). The topmost level (the HTML level) is formed by the content of the document modeled using the specification of HTML. This last level is probably the only one of interest for the user. Diff algorithms, however, rarely work at the topmost abstraction level; most of the times documents are compared at their bitstream level (e.g., binary diff), at the textual level (e.g., in versioning systems) or at the first non-textual abstraction level (e.g., XML). While all these comparisons are equally valid and correct, the lower the compared abstraction level is, the less meaningful the produced deltas are bound to be.

At each abstraction level different there are different kinds of elements that can be used to store the document knowledge. At the textual level there are characters, that can be binary octets, as is the case in ASCII, or longer strings of variable length, as is the case in UTF-8; at the XML level there are nodes, comments, processing instruction, etc. The whole content of a document at a certain abstraction level is fully described by the set of its elements and the set of relations between them. The elements carry the data of the document, the element relations store additional information about the elements, for example their order in the document or the way they are nested in each other.

6.1 Documents and abstraction levels

Definition 1 (Document)**.** A *document* (D) is a finite unit of stored knowledge, whose content is available at various levels of abstraction (D_{L_n}).

$$D = (D_{L_0}, D_{L_1}, \ldots, D_{L_n})$$

Definition 2 (Level of abstraction)**.** A *level of abstraction* (D_{L_n}) is a view on a document where the stored information is seen as a group of elements (E_n) and relations between elements (R_n) created according to a certain model (M).

$$D_{L_n} \equiv (E_n, R_n, M)$$

Documents must be finite in size, otherwise they would not be processable, at least not in practical terms. Streamed documents may be seen as documents of infinite size, at least in theory. Streamed documents are, however, internally processed in chunks; practically speaking these chunks are the individual documents upon which the programs operate.

A peculiarity of almost all electronic documents is that they have a single representation in terms of bits but multiple interpretations. For example, current word processing documents, e.g. ODT files, are, at the same time:

- a series of binary octets,
- a series of Unicode codepoints (i.e. text characters) encoded in UTF-8,
- an XML tree,
- an ODT document.

These different views on the same file can be seen as a stack of abstraction levels, shown in figure 6.1.

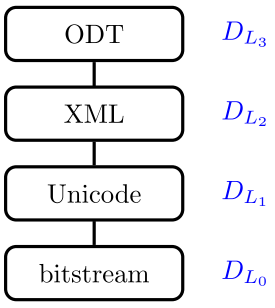

Figure 6.1: Stacked abstraction levels in ODT files

Definition 3 (Bitstream level)**.** The lowest level of abstraction (D_{L_0}) is the *bitstream level* where where the knowledge is represented as a series of binary digits ordered according the constraints imposed by the employed conceptual model and serialization format.

Definition 4 (File)**.** A *file* is the representation of a document at its bitstream level.

The lowest level of any electronic document is always the bitstream level, where all the information is encoded as a series of bits. Although very raw, this level is the one at which most of the text-based diff algorithms and all of the binary diff algorithms work. Algorithms that work at this abstraction level generate deltas that produce bit-perfect copies of the target document once applied, something that not all the algorithms working at higher level can guarantee.

Definition 5 (Model)**.** A *model* is a specification that indicates how the pieces of information are to be thought of and how they relate to each other.

Definition 6 (Format)**.** A *format* is a set of rules that specify how a document (D_1) modeled according to a model (M_1) can be transformed into a document (D_2) based on another model (M_2), usually a simpler lower-level model.

$$F_{M_1,M_2}(D_1) = D_2$$

The first step in shaping information into an electronic document is the choice of a model. Take, for example, a chapter in a book. In the HTML 4 model [61] a chapter starts when an element “h2” is used and its associated paragraphs are all the “p” elements that appear before the next “h2”. In contrast, in the DocBook 5 model [68] a chapter is identified by a “chapter” element and its paragraphs are all the “para” elements enclosed in it. The model used in a document is seldom chosen directly by the user, most of the times it is chosen by the application used to create the document.

Once the information has been assembled in a structure that fits the chosen model, it must be serialized according to the rules of its format in order to be stored in a file. The difference between a model and a format is that the model prescribes abstract data structures and connection between, the format, instead, describes how to store these data structures into bits or elements of the underlying model. To continue with the previous example, the format used to serialize DocBook trees is XML, the DocBook trees are thus written as XML trees. The conversion is quite straightforward because DocBook has been designed from the beginning as an XML-based format. Once the DocBook XML trees have been generated, they are serialized as sequences of characters following the rules of the XML serialization rules. The serialization process is not over yet. The serialization rules of XML produce sequences of Unicode codepoints, not bytes. To get a sequence of bytes we must serialize these codepoints into bits, for instance using the UTF-8 format. To recap: the serialization of a DocBook document goes through these steps (depicted in figure 6.2):

1. the application models the user content as a DocBook structure;

2. the DocBook structure is turned into an XML document according to the DocBook specs;

3. the XML document is turned into a sequence of Unicode codepoints according to the XML production rules;

4. the sequence of Unicode points is turned into a sequence of bits according to the UTF-8 rules.

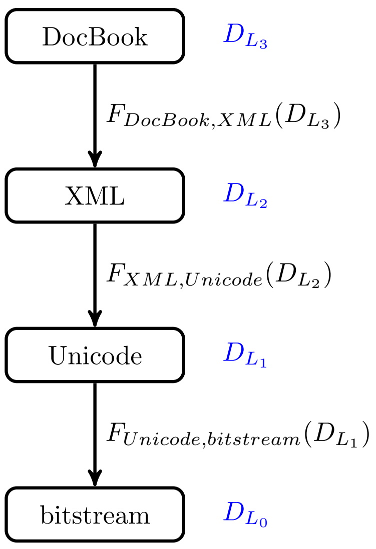
DocBook
D_{L_3}
$F_{DocBook,XML}(D_{L_3})$
XML
D_{L_2}
$F_{XML,Unicode}(D_{L_2})$
Unicode
D_{L_1}
$F_{Unicode,bitstream}(D_{L_1})$
bitstream
D_{L_0}

Figure 6.2: Serialization steps for a DocBook document

Not all the stacks of abstraction levels in a document are linear. Given a file there is only one possible stack of abstraction level. The converse does not hold: given a more abstract level, there may be different sub-stacks leading to different bitstream representations. This is the case for models that can be serialized using different syntaxes. Examples of this are RDF, that can be serialized using XML, N3 or other syntaxes, or the meta-models such as EARMARK/FRETTA [7] that can be used to encode any hierarchical document into a set of RDF statements. Yet another example is XML Schema that can be serialized using the canonical XML syntax or an alternative syntax such as XDTD [8]. Figure 6.3 illustrates the different possible stacks of abstraction level for an XML Schema document using two different syntaxes for a given XML Schema.

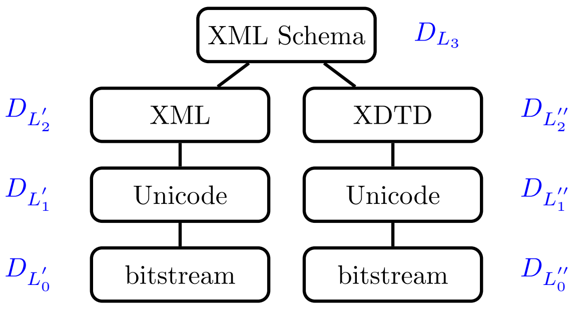

Figure 6.3: Possible different abstraction level stacks for XML Schema

6.2 Elements and element relations

At each abstraction level the content of the document is stored in elements. Depending on the abstraction level, the document will be seen as a composition of different types of elements: bits at the bitstream level, characters at the ASCII level, nodes at the XML level. At certain levels more than one type of elements can be found: for example, at the XML level the possible element types are nodes, processing instruction, comments, characters, entity references, etc.

The set of elements by itself is not enough to represent the content of a document in its entirety. The other needed piece of information is the relation between elements: what is the sequence in which these elements appears? are they nested? The element relations are used to store information about the way the elements are structured in a document at a certain abstraction level.

Definition 7 (Element)**.** An *element* (e) represents the smallest addressable unit of content in a document at a certain abstraction level. Each element is composed of data (D), a type (T) and an univoque identified (ID).

$$e \equiv (D, T, ID)$$

Definition 8 (Element relation)**.** An *element relation* r describes the relation t that exists between the element a and the element b.

$$r \equiv (t, a, b)$$

The selection operators for the relation r are $r.Type$, $r.A$ and $r.B$.

$$\begin{aligned} r.Type =& t \\ r.A =& a \\ r.B =& b \end{aligned}$$

Definition 9 (Sequence relation)**.** A *sequence relation* r indicates that element a follows element b under a strict total order relation.

$$r \equiv (\text{:sequence}, a, b)$$

The selection operators for all the sequence relations is $R_{sequence}$.

$$R_{sequence} \equiv \{r \in R : r.Type = \text{:sequence}\}$$

Definition 10 (Containment relation)**.** A *containment relation* r indicates that the element a contains the element b.

$$r \equiv (\text{:contains}, a, b)$$

The selection operators for all the containment relations is $R_{containment}$.

$$R_{containment} \equiv \{r \in R : r.Type = \text{:contains}\}$$

Definition 11 (Reference relation)**.** A *reference relation* r indicates that element a makes a references to element b.

$$\begin{gathered} r \equiv (t, a, b) \\ \text{where } t \notin \{\text{:sequence}, \text{:contains}\} \end{gathered}$$

Element relations are used to describe the fact that certain elements contain references to other elements.

An example of element relations can be found in RDF documents. Each statement with an object predicate has as its object another RDF entity. In practical terms the RDF statement “`#jack likes #jane`” is formalized with the document D_r shown in figure 6.4.

$$\begin{aligned} e_1 &= (\#\text{jack}, \text{RDF-entity}, \text{RDF-ID}\,(\#\text{jack})) \\ e_2 &= (\#\text{jane}, \text{RDF-entity}, \text{RDF-ID}\,(\#\text{jane})) \\ E &= \{e_1, e_2\} \\ R &= \{(\text{likes}, e_1, e_2)\} \\ D_r &= (E, R) \end{aligned}$$

Figure 6.4: Element relations for RDF statement “`#jack likes #jane`”

A similar situation occurs in documents that contain separate logical blocks with links between elements of these blocks. For instance, the document format for legal documents Akoma Ntoso [4, 3, 6] employs such a structure in its documents. The content of the document is strictly divided in different blocks (each with its own logical model and authoriality information) and the elements of the logically more abstract layers refers to pieces of the more concrete layers via local identifiers. For instance, the legal analysis of the outcome of a judgment, contained in the `judicial` block, refer to the exact words used to justify the judge sentence via the XML IDs of the elements in the `body` block.

6.3 Document structure

While all the documents are linear in the bitstream form, at more abstract level the information is often modeled in non-linear ways. For example XML documents are seen as particular hierarchical documents and RDF knowledge bases as graphs of linked resources. There are various special cases of document structure that can be identified.

The most general kind of document structure is the graph structure, as it does not impose any limit on the kind of relations that exist between the elements of the document. The hierarchical structure, instead, requires that only containment and sequence relations are used, thus limiting the ability to create arbitrary links between elements. Last, the linear structure impose that not even containment relations be used, only sequence relations are allowed and they also need to form a total order over the elements.

Definition 12 (Graphical document)**.** A *graphical document* is a documents whose elements are linked by relations of any kind.

Definition 13 (Hierarchical document)**.** A *hierarchical document* (or *tree document*) is a document where elements are linked using only containment and sequence relations.

$$IsDocHierarchical\,(D) \equiv D.R = D.R_{order} \cup D.R_{containment}$$

Definition 14 (Linear document)**.** A *linear document* is a document whose elements are linked using only sequence relations and the sequence relations define a total order over the elements of the document.

$$\begin{aligned} IsDocLinear\,(D) \equiv & D.R = D.R_{order} \wedge \\ & AllElementOrdered\,(D.E, D.R) \end{aligned}$$

6.4 Equivalence between documents and comparability

As explained in chapter 5, the key concept used by diff algorithms is that of equality between parts of documents. What is equal in both documents is aligned, what is different is analyzed and reported as the difference between the two documents.

The concept of equality can, in most cases be substituted with that of equivalence, i.e. equality modulo some details in the format used to store the document. For example, the two XSL statements in figure 6.5 are equivalent at the XSL abstraction level (both declare a variable whose value is the content of `doc//info/@title`) but not at the XML abstraction level nor at any other lower level.

Listing 6.1: Inline variable value selection

```
<xsl:variable name="title"
              select="$doc//info/@title" />
```

Listing 6.2: Nested variable value selection

```
<xsl:variable name="title">
    <xsl:value-of select="$doc//info/@title" />
</xsl:variable>
```

Figure 6.5: Two equivalent XSL documents

The concept of equality must be defined for each element type of an abstraction level. Without their definition, it would not be possible to compare documents at that abstraction level.

Definition 15 (Equality relation)**.** An *equality predicate* eq_L at level L returns whether the *equality relation* for that abstraction level holds for the compared two documents or elements.

Definition 16 (Equivalence)**.** For each equivalence predicate and relation there is an associated pair of equivalence modulo format predicates and relations that performs a comparison taking into account only the model of that abstraction level, not its format. An *equivalence predicate* returns whether the *equivalence relation* for that abstraction level holds for the compared two documents.

The point of using difference algorithms is finding parts of documents that differs, i.e. that are not the "same". The concept of difference relies thus on that of equivalence: without an equivalence relation between documents or parts of document it would not be possible to highlight what is different.

The simplest equivalence relation one can think of is the bitstream-level equivalence, it is also universal because all the electronic documents must have a bitstream representation. However, comparisons at the bitstream level, the so called binary comparisons, produce low quality deltas in almost all cases. Knowledge of the used format allow the production of deltas that are both smaller and more readable. In fact, many comparisons are higher text comparisons or format specific comparison. This means that more equivalence relations are needed, one for each model.

Another aspect that must be taken into account when comparing documents is that some of their abstraction levels may not be comparable, i.e. there is no equivalence relation defined between the models used at the same abstraction layer.

Take, for example, the abstraction levels of two OWL ontologies, the first saved using the RDF/XML serialization, the second using the RDF/N3 serialization, illustrated in figure 6.6.

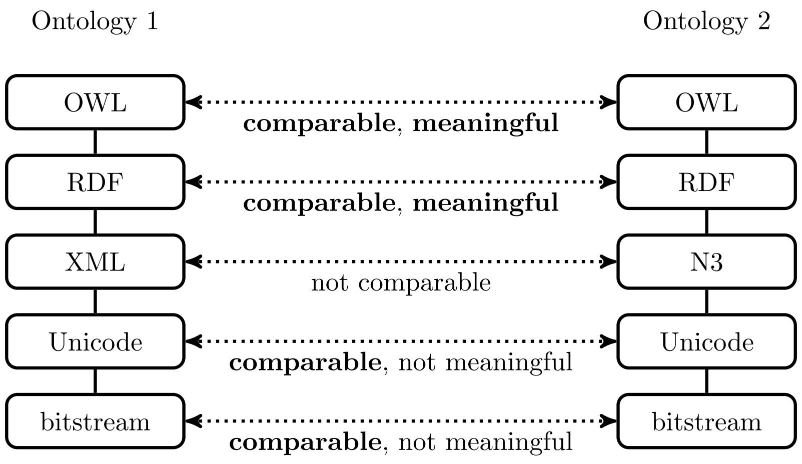
Ontology 1
Ontology 2
OWL
RDF
XML
Unicode
bitstream
OWL
RDF
N3
Unicode
bitstream
comparable, **meaningful**
comparable, **meaningful**
not comparable
comparable, not meaningful
comparable, not meaningful

Figure 6.6: Abstraction levels for two OWL ontologies in different serialization

In this case there are levels at which is sensible to compare, levels where it is not possible to compare and levels where a comparison is possible but with probable poor results. At the topmost abstraction level, both files are seen as collection of OWL axioms. These axioms are hard to compare because finding equivalent axioms is not a computationally trivial task [29]; on the other hand, a comparison between these two levels would probably produce an high quality result. A comparison at the second level is possible because the RDF spec provide a way to identify equivalent axioms in an RDF document. On the contrary, a comparison between an XML document and an RDF document is not possible as there is equality relation defined for this combination of models. Comparison at the lowest level is possible as both models are based on Unicode and produce a bitstream in the end. However, these comparisons will hardly produce anything useful: the most probable delta that can be produced at these levels is a trivial delta in which all the content of the source document is removed and replaced with all the content of the second document. This phenomenon appears also when the compared files are two different serializations of the exact same ontology.

Chapter 7

Deltas

The purpose of a diff algorithm is to find a list of differences that exist between two documents. A delta (section 7.1) represents this list of differences. In its most basic form a delta is just a container of found changes, usually a list. However, more advanced algorithms do not just produce plain lists of changes, for instance Papavassiliou et al. [38] aggregates changes in other changes and these aggregation relations must be recorded somewhere. The change relations (section 7.2) are the objects used to store information about these aggregations and about other relations between changes, for example the order in which the changes must be applied by a patching tool. Related to the application order relations is the concept of edit script. An edit script (definition 21) is a particular kind of delta where all the changes are connected by an application order relation, forming a strictly ordered chain of changes. Another particular kind of delta are the reversible deltas (definition 22): deltas that contain enough information that can also be used not only to transform the source document into the target document, but also the other way around: they can be used to transform the target document into the source document.

7.1 Deltas

The output of a diff algorithm is a so called delta. A delta is a collection of changes and change relations. The changes stored in the delta are the changes detected by the algorithm. The change relations are objects used to describe various relations that may exist between changes; for example, an application order relation between c_1 and c_2 states that the change c_1 must be applied before c_2 is applied in order for the delta to be correct.

Definition 17 (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).

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

Definition 18 (Empty delta)**.** An *empty delta* is a delta that contains no changes.

$$\Delta_{empty} \equiv (\emptyset, \emptyset)$$

Deltas are used to group together the changes found by an 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 an algorithm during its computations.

The delta as defined here is only a conceptual modelization of the results of a diff algorithm. In order to be used in practice, deltas must be serialized, either into concrete data structures, in case the algorithm is used as a library inside a bigger application, or as a file, in case the delta is to be stored or transmitted to external applications. Various possible serializations, suitable for different environments, are described in chapter 12.

7.2 Change relations

The changes by themselves are not enough to constitute a working delta. 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). Another useful information about the changes is whether they have been assembled starting from other changes or if they have been natively detected. All these pieces of information are recorded in the delta using change relations (the set R in definition 23).

In general terms, change relations are objects used to record that a certain relation exists between a certain changes. The meaning and the intended effects of a change relation are described by the kind of that relation.

Definition 19 (Change relation)**.** A *change relation* is a tuple describing the fact that there exist a relation of type K between the changes C_1 and the set of changes C_2.

$$r = (K, C_1, C_2)$$
$$\text{where } K \text{is the relation kind, } C_1, C_2 \subset C$$

Not all the changes present in a delta have the same role. Some are meant to be read by patching tools to reconstruct the modified document, other changes are used to justify the presence in the delta of other more sophisticated changes. The relations between the found changes are stored in R, the set of change relations.

Change relations can be used to store relations between sets of changes and other sets of changes (e.g., “changes 3, 7, 89 must all be applied before changes 4, 12, 72”), but more commonly they put in relation a single change with a set of changes (e.g., “change 34 encapsulates changes 12, 13, 21”).

Definition 20 (Application order relation)**.** An *application order relation* is a kind of change relation ($K = application\,order$) that is used to define the order in which changes are meant to be used. $R_{application\,order}$ is the set of all the application order relations.

$$R_{application\ order} \equiv \{r \mid r \in R, r.K = application\ order\}$$

The application order relation is the most basic type of change relations. A single application order relation describes what changes must be applied before applying what changes. Taken all together, the application order relations define a partial order over the changes.

7.3 Special deltas

There are cases in which algorithms produce deltas with particular properties. One such kind of deltas are the edit scripts: deltas where the order in which the changes must be applied is completely specified. Another special kind of deltas are the reversible deltas: deltas that are meant to be used to transform the source document into the target document but that, additionally, can be reversed so that the reversed delta can be used to recover the source document from the target document; the particularity of reversible deltas is that they contain enough information to make the reversing process feasible, a property not shared by all deltas.

Definition 21 (Edit script)**.** An *edit script* is a delta in which the application order is a strict total order defined for all changes in the delta.

Edit script are the most common kind of deltas produced by current implementations of the diff algorithms. What most implementations produce is a simple list of changes, to be applied one after the other in the specified order. Edit script have their origin in the way the first patches were produced, as small scripts for text editors. For instance, the original UNIX diff [23] produced a script written using the instruction of the UNIX qed editor. As such, these patches were small programming scripts rather than explicit recording of the changes between the compared documents.

Edit script have a main drawback: the fact that the tools that use them must follow the order in which the changes are applied. This means that the tools are not free to apply only certain parts of the delta, even if they are logically independent.

Definition 22 (Reversible delta)**.** A *reversible delta* is a delta where all the negative changes contain the same data that it would be needed in the opposite positive change.

In order for a delta to fulfill its intended purpose, it must contain enough information to run a copy of the source document into a copy of the target document. This means that the delta must carry all the data that is in the target document but not in the source document. There is, on the contrary, no need for a delta to carry any information about the data that is only in the source document but not in the target document, i.e. deleted data. For example, a delta could simply say "delete characters 7 to 12" instead of "delete characters 7 to 12: the word 'happy"'. However, if the delta has no information about what has been deleted, it will not be possible to invert the delta to generate another delta that turns the target document into the source document. A reversible delta contains all the information about the data present in the source but not in the target document, allowing the inversion of the delta itself.

Chapter 8

Changes

The differences found by a diff algorithm are expressed as a set of changes. Each of these changes (section 8.1) describes an operation that must be done on the source document to reconcile one of the found documents, in other words to make the source document more similar to the target document.

Some of these changes are considered atomic (definition 25) because they are found and generated by the algorithm looking only at the content of the source and target documents. Other changes, instead, have been generated by analyzing some of the changes that have already found. This happens for instance in the Hunt-McIllroy algorithm [23] where pairs of additions and deletions changes are aggregated into update changes or in Papavassiliou et. al [38] where similar pairs of basic changes on ontologies such as $DomainRemoved\,(prop, d_1)$ and $DomainAdded\,(prop, d_2)$ are aggregated into $DomainChange\,(prop, d_1, d_2)$. Changes of this latter kind, generated taking into account also other changes, are called complex changes (definition 26) and the link between the generated changes and the changes used to generate it are recorded through encapsulation relations (definition 24), a kind of change relation.

The generation of complex changes follows rules dictated by the algorithms, for example an update requires the presence of an addition and of a deletion on the same element. These changes that are used to justify the generation of a complex change are the main encapsulated changes (definition 28). All the other changes that are encapsulated in a complex change because they are allowed to be encapsulated but whose presence is not required by the rules are deemed additional encapsulated changes (definition 29).

Not all the complex changes are generated for the same reason. Some complex changes are used to express in a concise way that the same operation has been applied over adjacent elements. These complex changes are range changes (definition 30). Similarly, structural changes (definition 31) are changes that group other changes to make it explicit that the encapsulated changes follow a certain structure. An example of a structural change is the removal of a subtree in an XML tree: the removal change will encapsulate many other node-removal changes, mimicking the hierarchy found in the original XML document. Another kind of complex changes are meaningful change (definition 32): changes that encapsulate other changes in order to convey a certain meaning to that group of changes; for instance the meaningful change DOCBOOK-SECTION-SPLIT encapsulates at least the operations XML-ELEMENT-SPLIT and an XML-ELEMENT-ADD('title').

8.1 Changes

Definition 23 (Change)**.** A *change* is the application of an operation to the source document, or part of it, with the intent of reconciling a difference.

$$c \equiv (op, params)$$

Definition 24 (Encapsulation relation)**.** An *encapsulation relation* is a kind of change relation that links a container change to its encapsulated changes.

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

Encapsulation is a mechanism used by many algorithms to produce more meaningful deltas: once basic differences have been found, these differences are matched together producing a bigger change that encapsulates the matched smaller changes.

Definition 25 (Atomic change)**.** An *atomic change* is a change that does not encapsulate any other change.

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

For most algorithms, atomic changes are the only changes present in the generated deltas. The most common kinds of atomic operations are addition and removal changes.

Definition 26 (Complex change)**.** A *complex change* is a change that encapsulates at least one other change.

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

Complex changes are generated by algorithms during their refinement (or matching) phase: during that phase changes already detected are grouped according to the algorithm rules into more meaningful changes.

An example of a complex change generated from atomic changes is the UPDATE change. An UPDATE change is generated aggregating two atomic changes: an ADD change and a REMOVE change. Obviously not all pairs of ADD and REMOVE changes can be aggregated into UPDATE changes: the aggregation can happen only if certain conditions are fulfilled, in the case of UPDATE, the ADD and the REMOVE changes must both operate on elements that share the same position in the document. The concept of encapsulation justification is explained more in depth in section 8.2.

Definition 27 (Top level change)**.** A *top level change* is a change that has not been encapsulated in any other change.

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

8.2 Encapsulation justification

A complex change can be generated only if all the necessary other changes that contribute to its meaning are available. For example, the OWL-CLASS-ADDED(Person) change cannot be generated without the presence of a OWL-CLASS-DECLARED(Person) change. There are however other changes that could be encapsulated by a OWL-CLASS-ADDED(Person) change, but that, by themselves are not enough to justify the generation of that change. For example, a OWL-DOCUMENTATION-ADDED(Person) change could be encapsulated by a OWL-CLASS-ADDED(Person) change, but, on the contrary, the simple presence of a OWL-DOCUMENTATION-ADDED(Person) change is not enough to justify the generation of an OWL-CLASS-ADDED(Person) change.

Definition 28 (Main encapsulated changes)**.** The *main encapsulated changes* is the subset of encapsulated changes without which the generation of the encapsulating complex change could not be justified.

Definition 29 (Additional encapsulated changes)**.** The *additional encapsulated changes* is the subset of encapsulated changes that are not needed to justify the generation of the encapsulating complex change.

8.3 Classification of changes

Complex changes are generated for different purposes. These purposes can be classified in three groups: range changes, structural changes and meaningful changes.

The range changes are the complex changes that have been generated to group together a sequence of similar operations done to adjacent elements for example REMOVE-LINES(8, 12) groups together the changes REMOVE-LINE(8), REMOVE-LINE(9), ..., REMOVE-LINE(12). Structural changes are, instead, generated when a certain structure is recognized in a group of changes, for instance REMOVE-SUBTREE encapsulates various REMOVE-ELEMENT changes on elements that, together, form a proper subtree. Last, meaningful changes are generated to convey a certain meaning to a group of changes when taken together, for example HTML4-REMOVE-CHAPTER groups changes such as a HTML4-REMOVE-H2 and various HTML-REMOVE-PARAGRAPH.

The aim of an algorithm influences heavily the kind of complex changes it generates: algorithms that want to create concise deltas will support and detect range changes as they allow the generation of deltas with fewer top-level changes. Differently, algorithms that make an effort to find more advanced changes (e.g. NDiff [17]) or to generate domain-specific changes (e.g. PROMPTDiff [37] or OnEX [21]), will try to generate as many structural and meaningful changes as possible, using their knowledge of the domain to find meaning in the already generated changes.

Definition 30 (Range change)**.** A *range change* is a change that encapsulates similar changes made to a range of elements.

Definition 31 (Structural change)**.** A *structural change* is a change that encapsulates changes into a structure that resembles the structure of the elements in the document or the way the users made their modifications.

Definition 32 (Meaningful change)**.** A *meaningful change* is a change that encapsulates different changes with the purpose of providing a meaning to that group of changes.

Chapter 9

Objective properties of changes and deltas

The way changes and deltas have been defined in the previous chapters allows for the definition of objective properties that can be calculated using only the information present in the changes and in the delta.

These properties are not very interesting by themselves but they form the basis for more meaningful analyses such as those shown in chapter 14 (analysis of the qualities of deltas) and chapter 15 (detection of development phases of OWL ontologies).

9.1 Objective properties of changes

The objective properties of changes deal with two aspects of the changes as defined in the proposed model: the way changes are encapsulated (e.g., population, depth) and the data contained in them (e.g., modified elements, touched elements).

Definition 33 (Population of a change)**.** The *population of a change* is the total number of changes of which a change is composed of, including itself and the recursive closure of the encapsulated changes.

$$\mathit{Population}\,(c) = 1 + |\mathcal{E}_c|$$
$$\text{where } \mathcal{E}_c = \{e \mid (c,e) \in \mathcal{E} \vee \exists x_1, x_2, \ldots, x_n : (c,x_1), (x_1,x_2), \ldots, (x_n,e) \in \mathcal{E}\}$$

Definition 34 (Depth of a change)**.** The *depth of a change* is the maximum number of encapsulation layers that must be crossed to reach an atomic change.

$$Depth\,(c) = 1 + \max_{e \in \mathcal{E}} Depth\,(e)$$

Definition 35 (Width of a change)**.** The *width of a change* is the number of distinct changes encapsulated directly inside the change.

$$Width\,(c) = |\mathcal{E}_c|$$

Definition 36 (Touched elements of a change)**.** The *number of touched elements of a change* is the number of distinct pieces of information that are included as part of the change or of the encapsulated changes.

$$TouchedElements\,(c) = |\mathcal{D}_c|$$
$$\text{where } \mathcal{D}_c = \{d \mid (c, d) \in D\}$$

Definition 37 (Modified elements of a change)**.** The *number of modified elements of a change* is the minimum number of elements that must be modified by the change to fulfill its purpose. This number is always equal or less than TouchedElements.

$$ModifiedElements\,(c) =$$
$$\arg\min_n \left\{\phi\,(S, c_n) = \phi\,(S, c)\,, TouchedElements\,(c_n) = n\right\}$$
$$\text{where } \phi\,(D, c) \text{ is the application of } c \text{ to } D$$
$$\text{and } c_n \text{ is a change equivalent to } c$$

9.2 Objective properties of deltas

The objective properties defined for deltas are used to aggregate the homonymous properties of the enclosed changes (e.g., population, modified elements) or to provide an overview of the delta itself (e.g., number of top level changes).

Definition 38 (Population of a delta)**.** The *population of a delta* is sum of the Population property of all the top level changes.

$$Population\,(\delta) = \sum_{c \in TopLevel(\delta)} Population\,(c)$$

Definition 39 (Touched elements)**.** The *number of touched elements of a delta* is the sum of the NumberTouchedElements property of all the top level changes.

$$NumberTouchedElements\left(\delta\right) = \sum_{c \in TopLevel(\delta)} NumberTouchedElements\left(c\right)$$

Definition 40 (Modified elements of a delta)**.** The *number of modified elements of a delta* is the minimum number of distinct pieces of information that must be modified in order to turn S into T.

$$\begin{gathered} NumberModifiedElements\left(\delta\right) = \\ \arg\min_{n} \left\{\phi\left(S, \delta_n\right) = T, NumberTouchedElements\left(\delta_n\right) = n\right\} \\ \text{where } \phi\left(D, \delta\right) \text{ is the application of } \delta \text{ to } D \\ \text{and } \delta_n \text{ is a delta equivalent to } \delta \end{gathered}$$

Definition 41 (Number of top-level changes of a delta)**.** The *number of top-level changes of a delta* is the number of changes that are not encapsulated in any other change.

$$NumberOfTopLevel\left(\delta\right) = \left|TopLevel\left(\delta\right)\right|$$

Definition 42 (Separability degree of a delta)**.** The *separability degree of a delta* is number of maximally connected graphs (using changes as vertex and references to other changes as edges) that can be found in a delta.

$$Separability\left(\delta\right) = \left|\left\{\mathcal{I}_i\right\}\right|$$

$$\begin{aligned} \text{where} \quad \mathcal{I}_{\mathrm{i}} = \quad & \{c | c \in \delta.Changes, \\ & \forall rc : r \in R, r = \left(t, c, rc\right) \vee r = \left(t, rc, c\right), \\ & c \in \mathcal{I}_i, rc \in \mathcal{I}_i, rc \notin \mathcal{I}_{n \neq i}\} \end{aligned}$$

Deltas with an high separability degree can easily be split in smaller independent deltas, making it easier to isolate individual changes and to apply only a part of a delta.

Chapter 10

Operations

Once an algorithm has detected a difference, it records it as a change. The kind of the change is called its operation. For example, the change “ADD(’Hello’, ’32’)” has operation “ADD”. This means that the diff algorithm has detected a change that fits the description and the semantics of the ADD operation and that the data on which it has detected this change, i.e., the parameters of the operation, are the string “Hello” and the position index “32”.

In order for these operations to be understood by the tools that read the deltas, each operation must have an associated semantics. The semantics of the operations operates on three objects: the arguments of the operation, the content of the source document and the content of the target document. Using this information, the semantics of an operation describe how to transform a part of the source document into a part of the target document.

Not all algorithms detect the same set of operations. It is the task of the designers of an algorithm to decide which operations it should or should not detect. As discussed in section 3, it is quite common for algorithms to waive the ability to recognize many different operations to achieve a lower computational complexity.

10.1 Operations

An operation is a function that describes the effects of a change. While a change states that “something” must be done to a certain part of the document, an operation defines how to perform that “something”.

Definition 43 (Operation)**.** An *operation* is a function that takes a document (D) and a list of parameters (P) and returns a new document (D).

$$op : D \times P \to D$$

The application of an operation transforms a part of the source document into a part of the target document using a list of parameter to modify its behaviour.

10.2 Parameters

The parameters of an operation are used by the semantics of the operation itself to understand on which parts of the document the operation must operate and to supply data that is not present in the source document, for example elements that have been added and are present only in the target document. There are two types of parameters: pointer parameters (that are used to refer to a single element of a document, e.g. "the third byte" or "the first child of the forth child of the root node") and data parameters (that carry immediate data elements, e.g. a string of bytes or an XML node).

Definition 44 (List of parameters)**.** The *list of parameters* of an operation defines which pieces of data must be supplied for its application.

$$P = (p_1, p_2, \dots, p_n)$$

Definition 45 (Pointer parameter)**.** A *pointer parameter* is an operation parameter that refers to a document element through a pointer.

There two main kinds of pointer parameters: position pointers (for linear structures) and ID pointers (for graphs). Pointers to linear structures are simple 0-based integers that point to the position between two element, i.e. the pointer n points at the position between element n and $n + 1$. These pointers can be used only on linear documents (or linear portions of documents) for which an order is defined via sequence relations. For document where an order is not defined or for portions of documents for which an order is not defined (e.g. the attributes of XML documents) ID pointers must be used. An ID pointer id points to the element e for which the relation $e.ID = id$ holds.

Definition 46 (Data parameter)**.** A *data parameter* is an operation parameter that is composed of the actual data used to perform the operation.

10.3 Semantics, conditions and effects

The semantics of an operation define how the application of the operation together with the parameters stored in the corresponding change modifies the document it operates on. Given the definition of a document (see definition 1), the semantics of an operation is defined in terms of set operations, i.e. additions and removals of elements and relations from the sets of which a document is made of. Operations can be classified depending on how the elements and the relations of the document are manipulated by the semantics of the operation. The four classifications, or polarities, are: neutral operations, positive operations, negative operations and bipolar operations.

Definition 47 (Neutral operation)**.** A *neutral operation* is an operation that does not modify neither the set of elements, nor the set of relations of a document.

Definition 48 (Positive operation)**.** A *positive operation* is an operation that adds new elements or relations to the document but does not remove any existing element or relation.

Definition 49 (Negative operation)**.** A *negative operation* is an operation that removes some of the existing elements or relations from the document but does not add any new element or relation.

Definition 50 (Bipolar operation)**.** A *bipolar operation* is an operation that adds to the document new elements or relations and, at the same time, removes some of the existing element or relation.

10.4 Composition

The composition of operations is the mechanism through which it is possible to define complex operations in terms of simpler operations. This mechanism reflects the way atomic and complex changes work: a complex change aggregates atomic changes or other complex changes; the operation used by the aggregating complex change is a composition of the operations of the aggregated changes.

In practical terms, the composition of operations is obtained requiring the application of other operations as part of the semantics of the operation itself. These applications are done via the `apply` function.

Chapter 11

Catalog of operations

This chapter presents a catalog of operations to be used as operations in changes. The first part of the catalog describes the basic operations: addition and removal. On top of these basic operations, various kinds of complex operations are defined. The complex operations are stacked in layers: first structural changes are defined (e.g. TREE-REMOVE, for hierarchical documents), on top of these, format-specific changes are defined (e.g. XML-TREE-REMOVE, for XML documents), then domain-specific operations (e.g. DOCBOOK-SECTION-REMOVE, for DocBook documents), sub-domain-specific operations (e.g. DOCBOOK-INTRODUCTION-REMOVE, for DocBook documents written using a particular set of guidelines) and so on.

In deltas that are meant to be processed by tools different from those used to create the deltas, it is fundamental for the producer and the receiver of the delta to understand down to the smallest detail the operations used in the delta, otherwise the application of the delta would produce a document that is not identical to the desired target document.

11.1 Operations on trees

The operations described in this section operate on hierarchical documents as defined in definition 13.

tree-child-add

The tree-child-add operation is used to add an element to the ordered set of children of an element in a tree document.

Parameters

e (element) child to add

e_{parent} (element) parent element
p (position) position of added child

Conditions
$length\left(childrenOf\left(e_{parent}\right)\right) \leq p$

Effects

$$\begin{aligned} E' &= E \cup \{e\} \\ R' &= R \cup \{(\text{:contains}, e_{parent}, e)\} \\ &\qquad \cup \{(\text{:sequence}, e_{p-1}, e), (\text{:sequence}, e, e_p)\} \\ &\qquad \setminus \{(\text{:sequence}, e_{p-1}, e_p)\} \end{aligned}$$

tree-child-remove

The tree-child-remove operation is used to remove an element from the ordered set of children of an element in a tree document.

Parameters
e (element) child to remove

Conditions None

Effects
$e_{parent} = parentOf\left(e\right)$
$p = positionAmongSiblings\left(e\right)$

$$\begin{aligned} E' &= E \setminus \{e\} \\ R' &= R \setminus \{(\text{:contains}, e_{parent}, e)\} \\ &\qquad \setminus \{(\text{:sequence}, e_{p-1}, e), (\text{:sequence}, e, e_{p+1})\} \\ &\qquad \cup \{(\text{:sequence}, e_{p-1}, e_{p+1})\} \end{aligned}$$

tree-children-add

The tree-children-add operation is used to add a set of elements to the ordered set of children of an element in a tree document.

Parameters
$\{e_1, e_2, \ldots, e_n\}$ (set of elements) elements to be added
e_{parent} (element) parent element
p (position) position at which the elements will be added

Conditions

$length\,(childrenOf\,(e_{parent})) \leq p$

$AllElementOrdered\,(B, R)$

Effects

$$\begin{aligned}
e_{p-1} =& elementInPos\,(p-1)\\
e_p =& elementInPos\,(p)\\
E' =& E \cup \{e_1, e_2, \ldots, e_n\}\\
R' =& R \cup \{r : \forall e \in \{e_1, e_2, \ldots, e_n\}, (\text{:contains}, e_{parent}, e)\}\\
&\setminus \{(\text{:sequence}, e_{p-1}, e_p)\}\\
&\cup \{(\text{:sequence}, e_{p-1}, e_1), (\text{:sequence}, e_n, e_p)\}
\end{aligned}$$

tree-children-remove

The tree-child-add operation is used to remove a set of elements from the ordered set of children of an element in a tree document.

Parameters

$\{e_1, e_2, \ldots, e_n\}$ (set of elements) elements to be removed

Conditions None

Effects

$\forall e \in \{e_1, e_2, \ldots, e_n\}$, apply tree-remove-child (e)

tree-node-wrap

The tree-node-wrap operation is used to move a set of element inside a wrapper element.

Parameters

e_w (element) wrapper element

$B = \{b_1, b_2, \ldots, b_n\}$ (set of elements) wrapped elements

Conditions

$\forall b \in B : b \in E$

$AllElementOrdered\,(B, R)$

Effects

$E' = E \cup \{e_w\}$

$$\begin{aligned} R' = R &\setminus \{r \in R_{containment} : (r.A = e_w, r.B = b)\} \\ &\setminus \{r \in R_{order} : (r.A \notin B \vee r.B \notin B)\} \\ &\cup \{\forall b \in B : (\text{:contains}, e_w, b)\} \end{aligned}$$

tree-node-unwrap

The tree-node-wrap operation is used to remove an element and move its children to the parent of the remove element.

Parameters

e (element) the element to unwrap

Conditions

$hasParent\,(e)$

Effects

$$\begin{aligned} e_{parent} =& parentOf\,(e) \\ pos =& positionAmongSiblings\,(e) \end{aligned}$$

$$upgradedChildren = childrenOf(e)$$

$$\begin{aligned} &\text{apply tree-child-remove}\,(e) \\ &\text{apply tree-children-remove}\,(e) \\ &\text{apply tree-children-add}\,(upgradedChildren, e_{parent}, pos) \end{aligned}$$

11.2 Operations on lists

The operations described in this section operate on linear documents as defined in definition 14.

sequence element add

The sequence-element-add operation is used to add an element to a sequence.

Parameters

e (element) element to be added

p (position) position where the element will be added

Conditions

$length\,(D) \leq p$

Effects

$$\begin{aligned} E' &= E \cup \{e\} \\ e_p &= elementInPos\,(p) \\ e_{p-1} &= elementInPos\,(p-1) \\ R' &= R \setminus \{r \in R_{order} : r.B = e_p\} \\ &\quad \cup \{(\text{:contains}, e_{p-1}, e)\,(\text{:contains}, e, e_p)\} \end{aligned}$$

sequence element remove

The sequence-element-remove operation is used to remove an element to a sequence.

Parameters

e (element) element to be removed

Conditions

$e \in E$

Effects

$$\begin{aligned} E' &= E \setminus \{e\} \\ e_{p-1} &= elementInPos\,(pos\,(e) - 1) \\ e_{p+1} &= elementInPos\,(pos\,(e) + 1) \\ R' &= R \setminus \{(\text{:contains}, e_{p-1}, e)\,, (\text{:contains}, e, e_{p+1})\} \\ &\quad \cup \{(\text{:contains}, e_{p-1}, e_{p+1})\} \end{aligned}$$

11.3 Operations on XML trees

The following operations are used to manipulate XML trees. Differently from the normal trees, the XML elements can be parents of two different sets of children: ordered children (elements and nodes in general) and unordered children (attributes). For this reason some XML operations are specialized for attributes.

xml-child-add

The xml-child-add operation is used to add a child to the set of ordered children of an XML node.

Parameters

e (element) child to add

e_{parent} (element) parent element

p (position) position of added child

Conditions

$length\,(childrenOf\,(e_{parent})) \leq p$

Effects

apply tree-child-add (e, e_{parent}, p)

xml-child-remove

The xml-child-remove operation is used to remove a child from the set of ordered children of an XML node.

Parameters

e (element) child to remove

Conditions None

Effects

apply tree-child-remove (e)
$R_a = \{r : r \in R, r = (\text{:contains\_attribute}, e, a)\}$
$A = \{a \mid \exists r : r \in R_a, r.B = a\}$

$E' = E \setminus A$
$R' = R \setminus R_a$

xml-children-add

The xml-children-add operation is used to add a set of children to the set of ordered children of an XML node.

Parameters

$\{e_1, e_2, \ldots, e_n\}$ (set of elements) elements to be added
e_{parent} (element) parent element
p (position) position at which the elements will be added

Conditions None

Effects

apply tree-children-add $(\{e_1, e_2, \ldots, e_n\}, e_{parent}, p)$

xml-children-remove

The xml-children-add operation is used to remove a set of children from the set of ordered children of an XML node.

Parameters

$\{e_1, e_2, \ldots, e_n\}$ (set of elements) elements to be removed

Conditions None

Effects

apply tree-children-remove $(\{e_1, e_2, \ldots, e_n\})$

xml-attribute-add

The xml-attribute-add operation is used to add an attribute to the set of attributes of an XML node.

Parameters

a (element) attribute to be added
e (element) element to which the attribute will be added

Conditions None

Effects

$$E' = E \cup \{a\}$$
$$R' = R \cup \{(\text{:contains\_attribute}, e, a)\}$$

xml attribute remove

The xml-attribute-remove operation is used to remove an attribute from the set of attributes of an XML node.

Parameters

a (element) attribute to be removed

Conditions None

Effects

$$e = elementWithAttribute\,(a)$$
$$E' = E \setminus \{a\}$$
$$R' = R \setminus \{(\text{:contains\_attribute}, e, a)\}$$

xml-element-split

The xml-element-split operation is used to split an XML element in two. The second part of the content of the XML element is added to a newly created element of the same type and with the same data (e.g. the same element name for elements).

Parameters

e (element) element to split

p (position) position at which the content of the elements must be split

Conditions

$length\left(childrenOf\left(e\right)\right) \leq p$

Effects

$E_{after} = \{e_c : e_c \in childrenOf\left(e\right), positionAmongSiblings\left(e_c\right) \geq p\}$

apply xml-children-remove $(\mathrm{E_{after}})$

$e_{new} = (e.Data, e.Type, \text{new-XML-ID})$

$p_{new} = positionAmongSiblings\left(e\right)+1$

apply xml-child-add $(e_{new}, e_{parent}, p_{new})$

apply xml-children-add $(\mathrm{E_{after}}, \mathrm{e_{new}}, 0)$

11.4 Example of extension: Operations on DocBook documents

The following operations show how it is possible to define operations at higher abstraction level based on the composition of operations defined at the lower abstraction levels. In the case of DocBook, the underlying abstraction level is the XML level, whose operations are used to define the semantics of the domain-specific DocBook operations.

docbook-paragraph-split

The docbook-paragraph-split operation is used to split a paragraph (a `para` element) in two. After the docbook-paragraph-split operation is applied, the first part of the content of the paragraph remains in the original paragraph while the second part is put in a newly generated `para` element placed just after the original paragraph. An example of the effect of docbook-paragraph-split is shown in figure 11.1 (with some whitespace removed for clarity).

Parameters

$para$ (element) paragraph to be split

p (position) position inside $para$ where the paragraph will be split

Conditions None

Effects

apply xml-element-split $(para, p)$

Listing 11.1: Before

```
<sect1>
        <title>Introduction</title>
        <para>It was a dark and stormy night. The
        wind howled and twigs and leaves scuffled.</para>
</sect1>
```

Listing 11.2: After

```
<sect1>
        <title>Introduction</title>
        <para>It was a dark and stormy night.</para>
        <para>The wind howled and twigs and leaves
        scuffled.</para>
</sect1>
```

Figure 11.1: Effects of docbook-paragraph-split

docbook-section-split

The docbook-section-split operation is used to split a section (e.g., a `sect1` element) in two. After the docbook-section-split operation is applied, the first part of the content of the section remains in the original section while the second part is put in a newly generated section placed after the original section and titled with the supplied title. An example of the effect of docbook-section-split is shown in figure 11.2 (with some whitespace removed for clarity).

Parameters

sect (element) section to be split

p (position) position inside *sect* where the section will be split

title (element) the *title* element of the new section

Conditions None

Effects

apply xml-element-split $(sect, p)$

$sect_{new} = xmlElementAfter\,(sect)$

apply xml-children-add $(title, sect_{new}, 0)$

Listing 11.3: Before

```
<sect1>
        <title>Introduction</title>
        <para>It was a dark and stormy night.</para>
        <para>The wind howled and twigs and leaves
        scuffled.</para>
</sect1>
```

Listing 11.4: After

```
<sect1>
        <title>Introduction</title>
        <para>It was a dark and stormy night.</para>
</sect1>
<sect1>
    <title>First words</title>
        <para>The wind howled and twigs and leaves
        scuffled.</para>
</sect1>
```

Figure 11.2: Effects of docbook-section-split

Chapter 12

Serialization of deltas

The model presented in the previous sections is a purely conceptual model. In order to be used it needs to be implemented. This implementation takes various forms depending on the use it is meant to fulfill. For use inside the implementation of algorithms, it will take the form of data structures, usually classes and objects. Instead, when the delta is to be transmitted to other separate applications or devices, it will be serialized in a file whose content will be modeled following the rules of a format.

Section 12.1 shows a set of data structures illustrated using UML diagrams. These data structures are meant to be used directly inside diff algorithms or for objects exported through an API.

These serialization formats are the last piece needed to have a thriving ecosystem of tools: using one of these formats, tools can offer real interoperability between tools, not only at the conceptual level but also in practical terms.

12.1 Data structure model in UML

The following UML model provides a concrete way to represent changes and deltas as data structures to be used inside the implementation of the diff algorithms and of the tools that use them.

In brief, the UML model consists of three main classes: Change, Delta and Hunk. The Change class is an abstract class used as the basis for the classes of the objects that will record single changes. The Delta class is used to group together various changes without caring about the order in which they are intended to be applied or whether they are related to each other. In contrast with the Delta class, the Hunk class is used to group together independent sets of changes and of the order in which they are supposed to be applied.

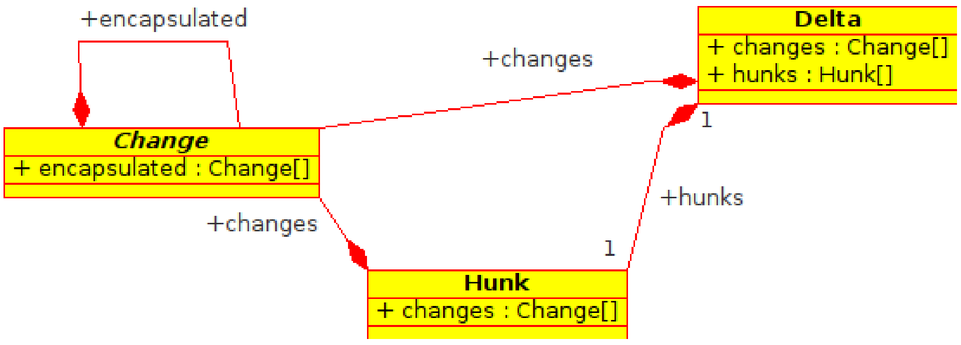

Figure 12.1: Overview of the classes

12.1.1 The `Change` class

Changes are represented as instances of the class Change. The methods of each change are used to access some of its relations either implicitly or explicitly: encapsulation relations are modeled through methods that return sets of changes, the application order relations are modeled externally, in the Delta and Hunk classes. The methods of the Change class are used to calculate the properties of each changes: population, depth, etc.

The Change class contains several virtual methods. These methods are declared as virtual because it is not possible to have a single way to compute their value, as it must be calculated in a different way depending on the type of change.

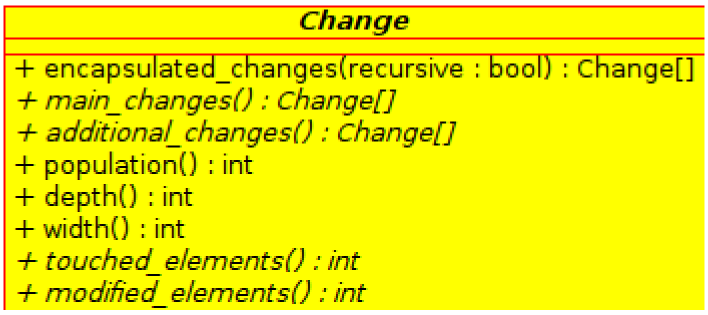
Change
+ encapsulated_changes(recursive : bool) : Change[]
+ main_changes() : Change[]
+ additional_changes() : Change[]
+ population() : int
+ depth() : int
+ width() : int
+ touched_elements() : int
+ modified_elements() : int

Figure 12.2: The `Change` class

12.1.1.1 `encapsulated_changes` method

The `encapsulated_changes` method returns the set of changes encapsulated by the change. The parameter `recursive` specifies whether the returned set should contain only the top level changes (`recursive = false`) or also recursive closure of all the encapsulated changes (`recursive = true`).

This method does not distinguish between necessary changes (without which the generation of the encapsulating change cannot be justified) and the additional changes (that can optionally be encapsulated but only once the necessary conditions for the generation of the change have been satisfied). Necessary and additional changes can be retrieved using, respectively, the `main_changes` method and the `additional_changes` method.

12.1.1.2 main_changes virtual method

The `main_changes` method returns the set of changes without which the generation of the change itself would not be justified, as explained in section 8.2.

12.1.1.3 additional_changes virtual method

The `additional_changes` method returns the set of changes that have been encapsulated with the change but that are not necessary for the generation of the change itself, as explained in section 8.2.

12.1.1.4 `population` method

The `population` method returns an integer with the value of the population property for the change, as defined in definition 33.

12.1.1.5 `depth` method

The `depth` method returns an integer with the value of the depth property for the change, as defined in definition 34.

12.1.1.6 `width` method

The `width` method returns an integer with the value of the width property for the change, as defined in definition 12.1.1.6.

12.1.1.7 `touched_elements` virtual method

The `touched_elements` virtual method returns an integer with the value of the touched elements property for the change, as defined in definition 34.

This method is virtual because the calculation of the number of touched elements differ from change to change.

12.1.1.8 `modified_elements` virtual method

The `modified_elements` virtual method returns an integer with the value of the modified elements property for the change, as defined in definition 34.

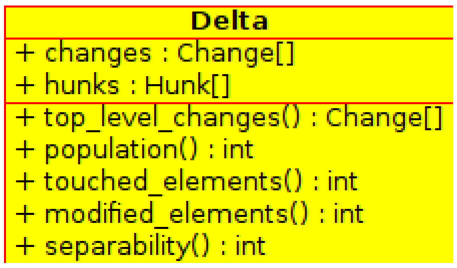
Delta
+ changes : Change[]
+ hunks : Hunk[]
+ top_level_changes() : Change[]
+ population() : int
+ touched_elements() : int
+ modified_elements() : int
+ separability() : int

Figure 12.3: The `Delta` class

12.1.2 The **`Delta`** class

The Delta class models the structure of a delta, described in chapter 7, and of its properties, described in chapter 9.

The Delta class is meant to be used to return the result of the computation of a diff algorithm but could also be used internally by the algorithm to keep track of the changes it has produced.

12.1.2.1 **`changes`** attribute

The `changes` attribute contains the set of all changes contained in the delta, i.e. the set C of definition 17.

12.1.2.2 **`hunks`** attribute

The `hunks` attribute contains the set of all hunks that compose the delta. Hunks and their intended use are described in 12.1.3.

12.1.2.3 **`top_level_changes`** method

The `top_level_changes` attribute contains the set of the top level changes contained in the delta, i.e. all the changes that have not been encapsulated in other changes.

12.1.2.4 **`population`** method

The `population` method returns an integer with the value of the population property for the delta, as defined in definition 38.

12.1.2.5 **`touched_elements`** method

The `touched_elements` method returns an integer with the value of the number of touched elements property for the delta, as defined in definition 39.

12.1.2.6 `modified_elements` method

The `modified_elements` method returns an integer with the value of the number of touched elements property for the delta, as defined in definition 40.

12.1.2.7 `separability_degree` method

The `separability_degree` method returns an integer with the value of the separability degree of the delta, as defined in definition 42.

12.1.3 The `Hunk` class

The Hunk class is used to store together changes that are meant to be applied together and in a certain order, similarly to the edit scripts as defined in definition 21. The existence of the Hunk class is justified by the fact that many algorithms produce edits scripts rather than generic deltas.

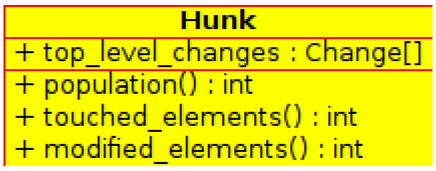
Hunk
+ top_level_changes : Change[]
+ population() : int
+ touched_elements() : int
+ modified_elements() : int

Figure 12.4: The `Hunk` class

The definition of the methods of the Hunk class correspond to the homonymous methods of the Delta class.

Part III

Applying and testing the universal delta model

Chapter 13

Practical applications of the model

The model presented in the previous chapters provides a solid foundation for practical application and significant analyses. The following chapters discuss how the presented model has been successfully applied in two different fields: the analysis of the qualities of the deltas produced by various algorithms and the development of a tool that is able to analyze the various phases of development through which an OWL ontology has gone through.

In section 14, the presented unified model is used to evaluate how fit an algorithm is for use in a certain environment for a certain task. This analysis is necessary because users rarely know which of the many different algorithms is the best for the problem they are trying to solve: the needs of a programmer are different from the needs of an editor that works with books in an XML format. This evaluation is based on a series of metrics calculates looking at the delta produced by an algorithm. The behavior of an algorithm (whether it creates redundant deltas, how precise or broad it is in the discovery of changes, its preference towards the creation of smaller or larger changes, etc.) is thus inferred from its output, not by looking at its code or its inner details.

Section 15 shows how the unified model helps the development of tools used to analyze copious amount of data. That section presents a tool that can be used to study the evolution of various documents, in this case of OWL ontologies. This tool employs a newly-developed algorithm to detect how a document has evolved, version after version. This analysis highlights the existence of various common phases during the development of ontologies, e.g. the “growth” phase, the “link to other ontologies” phase, the “documentation” phase, etc. The use of the unified model allows the same tool to be used to study other kinds of documents with few changes to the tool.

Chapter 14

Quality assessment of deltas and algorithms

This chapter presents a formal way to describe the quality (and qualities) of diff algorithms through the analysis of the produced deltas. Historically, designers of diff tools were mainly concerned about the computational complexity of the algorithms and the length of the produced edit scripts. Recently users started giving more relevance to the quality of deltas, designing tools that could produce better results: more readable, more usable, more natural. In contrast with the existing subjective way of evaluating algorithms, this section presents a multidimensional set of objective metrics to evaluate and compare the deltas produced by different algorithms to highlight what are their qualities and what are the most fit for use in a certain task.[1]

The automatic comparison of two different revisions of a document and the compilation of a list of changes that happened between those revisions are common tasks. The lists of changes, 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 and collaborating writers similarly signal changes they made to solicit comments; 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 and its imperfect copies.

Surprisingly enough, the quality of these deltas is not a key factor when designing and comparing diff tools. In fact, assuming that two algorithms are both correct - i.e. able to produce deltas that can be applied to the older document in order to

[1]These results have been extracted from [5].

obtain the newer one - they are mainly evaluated by comparing their performance. One of the reasons behind this fact is that historically such a research has been carried on by the database community, that has to deal with huge quantities of data and to reduce space and time consumption. It is not a case that 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.

The quality of an algorithm is often expressed in terms of its capability to reduce the size of the produced delta. As summarized in [15]: "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".

There is now a growing interest in characterizing more precisely the quality of deltas, in order to design algorithms that better fit the real needs of the users. For example, in [17] and [51] the authors strive to create a "better" delta, a delta that feels more "natural" to its users than a delta produced by a more canonical algorithm.

The focus of this chapter is, in fact, on measuring and comparing the quality of the deltas produced by diff algorithms. The chapter proposes a shift in the evaluation of such algorithms: instead of evaluating their execution process (for instance in terms of memory and time consumption), algorithms are compared by analyzing the output they produce.

This chapter introduces a framework for measuring the quality of these output through an objective evaluation process based on the properties defined in chapter 9. The basic idea consists of extracting numerical indicators from deltas (such as the number of detected changes, the number of touched elements, the number of high-level changes) 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 chapter is structured as follows. Section 14.1 describes more in depth the diff algorithms that focus less on speed or size and more on qualitative aspects. Section 14.2 discusses how the same concept of "quality" can have different meanings according to users' needs and preferences. Section 14.3 introduces our top-down solution, a set of aggregated metrics, in section 14.4. The application of the metrics is presented in section 14.5: 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 14.6.

14.1 Comparing quality of diff algorithms

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. In fact, a potentially infinite number of sequences of changes could be applied to build a document from another one.

There is a further tricky issue. As highlighted by [43] "*all approaches make use of different delta models, which makes it difficult to measure the quality of the resulting deltas*": not only the algorithms select different sequences of changes, but they even use their own internal model and recognize their own set of changes. Consider that some of them detect moves while others do not, or that the same name is used for different operations by different algorithms.

It is not a coincidence that most algorithms have been evaluated by only taking into account their computational complexity and execution time, and considering quality as a second-class dimension to evaluate.

This is particularly true in the context of XML diff, with few exceptions. In [17] 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 JNDiff, 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 shows lower performance (but still acceptable) in comparison to faster algorithms.

The idea of looking for deltas that better describe operations on literary documents has also been investigated by DocTreeDiff [43, 41]. The authors analyzed patterns in editing office documents and extracted rules for their identification. In particular, they studied the mapping between high-level changes and consequences on the underlying XML tree. The quality of the result of DocTreeDiff, in comparison to other algorithms, was measured by counting the number of edits contained in the deltas. 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 preliminar but shows a great variability of types of changes on the experimental data set, with evident differences between the results of different algorithms running on different documents. Such a variability could be an indicator of the ability of an algorithm to detect a larger and more precise set of changes.

The measurement of the number of edits is a quite common solution, experimented also by Xandy [31] and X-Diff [48]. This kind of comparison requires to study the internal structure of the delta. Nonetheless, in the context of XML diff where deltas are usually serializable into XML files, that is very easy through simple XPath expressions. The fact that each algorithm recognizes a different set of operations still remain an open issue. The absolute number of edits, in fact, is heavily influenced by the set of available operations and the way each algorithm combines them.

To solve this issue, many XML diff algorithms relate the quality to the size of the deltas they produce. This is the approach of XyDiff [15], whose evaluation has confirmed that the delta size is not optimized but the efficiency is very high. Even if not precise, as it does not investigate the nature of changes and their internal relationships, such evaluation can be fully automated and makes it possible to compare heterogeneous deltas.

A slightly different process has been proposed for measuring the quality of Faxma [32]. The authors compared 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 goes into details of the deltas but it reduces the noise generated by implementation choices of each algorithm. It is also interesting to notice that it has been proposed for an algorithm that does not produce an edit-script, i.e. a plain sequence of changes, rather a mix of references to unchanged content from the original document and newly inserted fragments. Such a format-independent evaluation is the easiest possible comparison between deltas intrinsically different.

A finer measurement has been proposed in the early days of tree-based diff algorithms, by [13]. They basically employed an edit cost model. 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 also introduced an algorithm that minimizes this cost. Such approach gives users the possibility of 'tuning' the results of the algorithm by giving different weights to each operation. The fact that costs are decided *a priori*, on the other hand, does not take into account other information that could be meaningful, for instance about the amount of nodes from the original document eventually stored in the delta.

This dimension has been recently used as quality indicator in [45]. This work is on detecting changes between unordered XML data by exploiting SQL queries instead of DOM-based representations and calculations. The quality of the presented algorithm is measured by measuring the ratio between the number of nodes in the produced delta and the number of nodes in the ideal one.

The idea of comparing the quality of deltas is gaining relevance for ontology diffing as well. This problem is even more complex than diffing tree-based documents.

One of the reasons is that ontologies contain a lot of implicit information that can be derived, for instance, by inheritance or inference. The fact that a diff algorithm takes or not this data into account has a great impact on the delta quality.

In [51] the authors introduced multiple differential functions to compute deltas, and argued that deltas cannot be considered correct in any context. In particular, they distinguished deltas containing only changes over explicit tuples and deltas working on inferred tuples too. Moreover they introduced some differential functions to calculate the smallest deltas and to accept/avoid redundancy in the final result. The paper also identifies some properties of the deltas such as reversibility, size minimality and redundancy elimination. These characterizations help designers to evaluate the quality of the delta produced by each approach and to decide which is the most appropriate for a given context.

Another tricky issue in diffing ontologies is that some changes might not be worth detecting as they are not effective. Consider for instance, the deletion of a triple that could be directly inferred by the others: should it be included in a delta? In which circumstances? In [20] authors discussed the importance of dealing with both effectual and ineffectual changes, in order to improve the quality of the delta. More important, they introduced a classification of changes that also takes into account ineffectual changes. The basic idea is that even those changes that do not affect the final set of (inferred) tuples are worth analyzing as they can provide users with information about the evolution of the ontology and/or about errors (for instance, detecting the duplication of an axiom).

There is a further tricky issue when detecting changes between ontologies. The most common operations on ontologies, in fact, generate a lot of small and scattered changes, that are worth interpreting and understanding as a whole. Consider for instance the introduction of a new class with some properties: a syntactical approach would produce a delta with a long list of changes (add class, add properties, add relations, etc.), while a more meaningful delta should recognize the higher-level operation. For these reasons, several ontology diff designers are striving for more accurate and flexible approaches to characterize deltas.

In [38] 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 more the delta contains high-level changes, the more its quality and effectiveness is high.

The work of [39] stressed on the importance of letting users decide which changes should be detected and how. 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). CDL is built on temporal logic and allows designers to express changes as differences between the current and the previous version of an ontology. Examples of changes that can be defined with CDL are: modification of the range of a property, introduction of a new subsumption relation, deletion of an object property, transformation of a data property into an object one, etc.

The aggregation of atomic changes into more complex structures that better capture the editing process has also been studied for XML database schema evolution [14]. The authors discussed a taxonomy of high-level changes and how each change can be expressed as combination of smaller units. The overall objective is to maximize the number of complex changes in order to improve the readibility and quality of the deltas. As stated by the same authors, however, the model is incomplete and does not cover all possible XML DTDs and schemas.

In conclusion, these definitions of "quality" have all been created to capture a single use of deltas, in very different contexts. We rather believe that is not enough to measure the quality of a delta as a single value. A better approach is to think of deltas as objects that have multiple evaluable dimensions, each able to capture one facet of quality. There are in fact contrasting needs and expectations in the same definition of such dimension.

14.2 Quality is domain-dependent

Users in different domains have different requests and expectations for a diff system. Not only these requests are different, often they also conflict with each other: for example in certain cases a cursory summary of what has changed is enough, in others an extreme level of detail is needed. These conflicting objectives are one of the reasons behind the existence of a myriad of specialized diff tools: each one is tuned for a specific activity. This heterogeneity makes it difficult to evaluate whether a certain diff system is suitable for use in a certain scenario.

There are many examples of how different algorithms can influence the fitness of the produced deltas in a particular environment. For example, there are two main algorithms used to produce diffs from two revisions of a source file: Hunt-McIlroy [23] and Patience [57]. Using lines as basic units of comparison, the first algorithm is able to create a very compact edit script that contains only the minimal number lines that has been added or removed. This compactness is a desirable property when the size of the generated patches matters, for example if they are used to store the history of that file. Differently, the Patience algorithm produces patches trying to lump together "local modifications", i.e. changes happened between blank lines or other similar conventions used to divide text files in sections. This mechanism attempts to reproduce the way changes are seen by programmers, although the patches produced using this technique are more verbose than those produced by Hunt-McIlroy.

The algorithm used to find the differences is not the only component that influences the perception of how good a delta is, the format in which the delta is stored or visualized also plays an important role. An example of this are two widespread formats: the contextual [66] and the unified [67] patch format . The contextual format shows for each change some lines common to both the source and target document, then it shows the modified part of the text, first how it was in the original document, then how it is in the modified one. The unified format, instead, shows only the modified document, highlighting lines that have been touched by the changes, but without showing what was present in those lines in the original version. We can say that the conceptual delta schema of the unified format defines only the ADD and DEL operations, while contextual format also generates UPDATE changes grouping together pairs of ADD and DEL changes that operate on the same line. It follows that the contextual format is more suitable for patches that are meant to be reviewed in detail, while the unified format offers a simpler representation that hides details not useful to who is only interested in seeing which parts of a document have been changed.

Another fact that impacts the quality of the deltas is whether the algorithm that generates them is specialized for use with certain formats. If an algorithm knows the inner details of a file format, it can easily produce more natural deltas. Take for example an XML document in which several attributes have been changed. When the original and the modified document are compared using a plain text diff, the produced delta will look like that in figure 14.1: correct but unnatural because full of unneeded details and repetitions like "line 6 changed from `<stanza id='first' tone-of-voice='whisper'>` to `<stanza id='intro' tone-of-voice='whisper'>`" or "characters $6 + 12 \ldots 6 + 16$ have been removed and `intro` has been added at offset $6 + 12$". On the contrary, the deltas produced by the XML tool will be more precise and concise, as they can say "the value of attribute `id` in node `/text/stanza[1]` has been changed from `first` to `intro`".

Nonetheless different XML tools might produce very different deltas, that are all correct but whose quality is very different. Consider for instance the simple case shown in figure 14.2: a bold style is added to a small fragment of a paragraph by wrapping it into a new element.

Delta A is useful to rebuild the modified document from the original one. On the other hand, it uses a lot of space and it is not precise. Delta B is more compact

```
--- a.xml
+++ b.xml
@@ -4,5 +4,5 @@
     <autor id="#ak9881jg2"/>
 </meta>
-<stanza id='first' tone-of-voice='whisper'>
+<stanza id='intro' tone-of-voice='whisper'>
     <verse>an so it is</verse>
     <verse>just like you said</verse>
```

Figure 14.1: XML files comparison with a pure-text tool

```
- original -
<p>A bold text.</p>

- modified -
<p>A <b>bold</b> text.</p>

- delta A -
<del><p>A bold text.</p></del>
<ins><p>A <b>bold</b> text.</p></ins>

- delta B -
<p>A <del>bold</del><ins><b>bold</b></ins> text.</p>

- delta C -
<p>A <b diff:wrap='wrap'>bold</b> text.</p>
```

Figure 14.2: Wrapping a text fragment in bold in XHTML

and precise but does not recognize exactly the editing action performed by the author, that is instead fully captured by delta C. Although simple, this example gives an idea of how much it is difficult to find, and even to define, which is the best delta when diff-ing tree-based documents.

14.3 A top-down approach to measure the quality of deltas

The previous discussion has shown that is very hard to find a conclusive definition for the "quality" of a delta. It is even more complex to translate such a dimension into one or more objective dimensions that can be evaluated automatically. To solve this issue we propose a top-down approach inspired by the Goal-Question-Metric approach (GQM) [9].

GQM is hierarchical measurement process. It starts defining 'Goals', i.e. purposes of the measurement: each goal, in turn, is characterized by specifying the object of the measurement (what is being evaluated?) and the viewpoint (who is interested in such evaluation? what for?). This last aspect is very important: the assumption is that any measurement cannot be disjoint by the users who will eventually perform it.

Like for software, the perceived quality of deltas is dependent from who is evaluating it. Thus, we first identify the goal of the evaluation by identifying the users are interested in it and the domain it will be applied to.

There are many goals that could be identified: e.g., “verify whether an algorithm is suitable for backups of big files”, “verify whether an algorithm is suitable for showing differences between two HTML files”, “verify whether an algorithm is suitable for direct use by coders”, “verify whether an algorithm is suitable for detecting changes between data-centric documents”, etc. However, this chapter does not set a predefined set of users and goals as it is expected that different domains will have different goals and only their users can specify them. Nonetheless, goals can be all summarized in one general meta-statement:

GOAL: Verify whether the algorithm fits user’s need

The second step of GQM deals with ‘Questions’. In this step each goal is broken down into several questions. The same approach can be applied to define and evaluate the quality of a delta. Consider for instance the following scenario: the goal of the user is to “verify if a diff algorithm [produces a delta that] is suitable for storing differences in a filesystem”. Sysadmin will be interested in some aspects of the output: they probably prefer a delta that does not waste space; they might also be interested in reading changes line-per-line, with a layout similar to the ones they are used to deal with; another requirements might be that the delta stores all information about deleted content (this is not always true, as some algorithms keep only references to the original resources). Thus, we need to formulate questions associated to these quality requirements that give sysadmin insights about the nature of the delta and the behavior of the algorithm that generated it.

Space consumption is definitely an interesting parameter to evaluate. It is actually been taken into account by several researchers in the past [64, 32]. Nonetheless, we believe there is a more precise evaluation besides the absolute measurement of the space used by the delta. It is interesting, in fact, to understand whether or not that space is actually required or the delta contains a lot of redundant information. The first question we propose is in fact:

Q1: Does the delta waste space?

It is not always true that the extra information included in a delta is useless.

In many case, it can be very useful for the final users to contextualize changes. The context is not strictly necessary but helps users understand what has being changed and where. This leads to a contrasting question:

Q2: How much information about the context of changes is stored in the delta?

Consider, for instance, the case of programmers that look for differences in source code. They expect the presence of a certain amount of context around the changed parts. New programmers 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. On the other hand, conciseness is an important parameter for the sysadmins that use deltas to only backup files (that will be primarily read by other applications, instead of human users).

Given that shorter deltas are easier to understand, another plausible question for evaluating the fitness of the deltas is:

Q3: How much is the delta summarized?

The answer to this question can be seen from two different angles: either the delta is short because it is composed of few changes (i.e., the algorithm tried hard to produced as few changes as possible) or because many changes have been grouped into composed changes (e.g., a pair of addition and deletion changes can be grouped in single update change).

Related to these aspects, there are two other questions that would be interesting to take into account:

Q4: Does the algorithm prefer simple or composite changes?

Q5: Is the algorithm able to identify domain-specific changes?

These last two questions are of interest for the users who have to understand the content of a delta, not just to apply it. For example developers of visualization tools need to analyze the content of a delta to create the corresponding graphical representation; for them having deltas that contain domain-specific changes may allow the generation of more precise and interesting visualizations.

The last step of GQM is the most important for our purposes. It consists of defining so called ‘Metrics’: quantifiable pieces information that are associated to every question in order to answer it in a quantitative way. We apply a similar approach: the basic idea consists of extracting quantitative indicators from a delta (such as the number of detected changes, the number of touched elements, the number of high-level changes) and aggregating them into more complex metrics (such as precision, meaningfulness) that can be combined with atomic indicators in order to give quantitative measurements about deltas. These parameters can be evaluated to answer the above-discussed questions and decide whether or not a delta achieves a given goal.

The introduction of the metrics requires a preliminary step, to make it possible to apply the same evaluation process to algorithms very different among each other. In fact, we need to establish a common conceptual schema on top of which metrics will be built. We present it in the following section, together with a small thesaurus of technical terms. Later, we will go into details of the metrics and their atomic building blocks (i.e. quantifiable properties of deltas).

14.4 Metrics for delta evaluation

Based on the quantifiable properties that we defined in the previous section, we can now define a series of metrics that represent the many dimensions among which it is possible to evaluate the deltas. These metrics are meant to capture quantitative data that can answer questions like those introduced in Section 14.3.

Differently from the previous properties, the selection and the definition of these metrics are domain-oriented. We expect to find new ways of combining the same atomic indicators to answer new questions in the future. In fact, the set of goals and questions in our model is open.

Like with properties, for each metric M there is a specialized version M_{tp} that uses the property $\mathrm{Prop}_{\mathrm{tp}}$ instead of Prop.

To simplify the explanation of these metrics we will base our examples on the two XML documents shown in figure 14.3 and some possible deltas generated by algorithms with different characteristics, shown in figure 14.4.

14.4.1 Precision

Precision indicates how many non modified elements have been included in the delta. Values of precision near 1 indicate that the delta contains almost only information about the occurred modifications; values near 0 indicate that almost all the information carried in the delta is redundant.

$$Precision\,(\delta) = \frac{\text{modified-elements}\,(\delta)}{\text{touched-elements}\,(\delta)}$$

Some algorithms produce deltas that contain redundant information in order to make the delta more readable while other prefer to avoid redundancy as much

Listing 14.1: Source document

```
<info>
        <author>John Doe</author>
</info>
```

Listing 14.2: Target document

```
<info>
        <author>
                <name>John</name>
                <surname>Doe</surname>
        </author>
</info>
```

Figure 14.3: Example modified XML document

as possible, focusing on reducing the length of the delta.

There are cases where extreme precision is required, for example when transmission bandwidth or storage space is scarce. In these cases every repeated byte is a wasted byte. Some algorithms go to great lengths to produce very precise deltas [64]. On the other hand, deltas that are too precise 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. This metric is an important indicator to answer, for example, the questions Q1 and Q2 shown in section 14.3.

The tradeoff represented by the Precision metric can be seen clearly comparing two possible ways to express the modifications made to the author element in the example, illustrated in figure 14.4. The first delta removes the subtree rooted on author and adds the new version of the same, the second delta wraps some characters with elements. For both deltas the number of modified elements is 2: the number of elements that must be added. The number of touched elements is, instead, different for each delta: 19 (4 elements and 15 characters) for the first delta and 11 (2 elements and 9 characters) for the second one. The precision of the first delta is thus lower than that of the second: 0.10 (i.e. $\frac{2}{19}$) versus 0.18 (i.e. $\frac{2}{11}$).

14.4.2 Conciseness

Conciseness indicates how much the changes found in the delta have been grouped into bigger changes. A very concise delta is a delta whose number of top-level changes has been reduced through the use of the various encapsulation mechanisms.

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

Listing 14.3: Delta 1

```
A.1: REMOVE-TEXT(John Doe)
A.2: REMOVE-ELEM(<author/>)
A.3: ADD-ELEM(<author/>, <book/>)
A.4: ADD-ELEM(<name/>, <author/>)
A.5: ADD-TEXT(John, <name/>)
A.6: ADD-ELEM(<surname/>, <author/>)
A.7: ADD-TEXT(Doe, <surname/>)
```

Listing 14.4: Delta 2

```
B.1: WRAP(John, <name/>)
B.2: WARP(Doe, <surname/>)
```

Listing 14.5: Delta 3

```
C.1: REMOVE-TEXT(John Doe)
C.2: ADD-TEXT-ELEMS(<name>John</name><surname>Doe</surname>,
                    <author/>)
    C.2.1: ADD-TEXT-ELEM(<name>John</name>, <author/>)
        C.2.1.1: ADD-ELEM(<name/>, <author/>)
        C.2.1.2: ADD-TEXT(John, <name/>)
    C.2.2: ADD-TEXT-ELEM(<surname>Doe</surname>, <author/>)
        C.2.1.1: ADD-ELEM(<surname/>, <author/>)
        C.2.1.2: ADD-TEXT(Doe, <surname/>)
```

Figure 14.4: Possible deltas for <author>

Algorithms can reduce the number of changes needed to express the modifications by grouping small changes into bigger changes, sometimes at the expenses of clarity or redundancy.

Question Q3 in section 14.3 can be answered by using this metric.

The amount of conciseness for the deltas shown in figure 14.4 can be calculated if the encapsulated changes are taken into account. The figure 14.5 illustrates how the shown changes encapsulate, in fact, smaller changes from which they have been deduced. The conciseness of the first delta ($1 - 7/7 = 0$) is smaller than the conciseness of the second delta ($1 - 2/9 = 0.778$). This reflects the fact that the second delta has been made more concise by the detection of complex changes.

14.4.3 Meaningfulness

Meaningfulness indicates how much of the delta conciseness is due to the use of complex changes. An high meaningfulness score indicates that the algorithm has been able to express much of what has been changed using meaningful changes,

```
B.1: WRAP(John, <name/>)
    B.1.1: REMOVE-TEXT(John Doe)
    B.1.2: ADD-TEXT-ELEM(<name>John</name>,
                          <author/>)
        B.1.2.1: ADD-ELEM(<name/>, <author/>)
        B.1.2.2: ADD-TEXT(John, <name/>)
B.2: WARP(Doe, <surname/>)
    B.2.1: = B.1.1
    B.2.2: ADD-TEXT-ELEM(<surname>Doe</surname>,
                          <author/>)
        B.1.2.1: ADD-ELEM(<surname/>, <author/>)
        B.1.2.2: ADD-TEXT(Doe, <surname/>)
```

Figure 14.5: Possible delta for <author> (delta 2 expanded)

going beyond the simple detection of atomic changes.

$$Meaningfulness\,(\delta) = \frac{\text{number-top-level}_{complex}\,(\delta)}{\text{number-top-level}\,(\delta)}$$

This metric, combined with the next one, can be used to give a quantitative answer to the questions Q4 and Q5 in section 14.3.

The examples in figure 14.4 show two very different levels of abstraction: the first delta is composed of atomic changes only so its meaningfullness values is 0. On the contrary, in the second delta all the atomic changes have been grouped into complex Wrap changes, making its meaningfullness value 1.

14.4.4 Aggregation

Aggregation indicates how much of the inner parts of the delta, not only of its topmost level, is expressed using complex changes instead of atomic changes. The better suited a conceptual diff schema is, the more likely an algorithm is to aggregate atomic changes into complex changes; an algorithm that know HTML operations will be able to detect more complex changes than an algorithm limited to XML operations or, even, to pure text.

$$\begin{aligned} Aggregation\,(\delta) &= \frac{\text{number-atomic-in-complex}\,(\delta)}{\text{population}_{atomic}\,(\delta)} \\ &= 1 - \frac{\text{number-top-level}_{atomic}\,(\delta)}{\text{population}_{atomic}\,(\delta)} \end{aligned}$$

Together with the previous metric, the value of Aggregation can be used to answer the questions Q4 and Q5 in section 14.3.

While this metric is positively correlated to abstraction, it focus on measuring *how much* has been aggregated in complex changes rather than *whether* complex changes have been used to express the most superficial level of the delta.

The three examples deltas in figure 14.4 show three different levels of aggregation: the first delta has not been aggregated at all ($1 - 7/7 = 0$); in the second all the atomic changes have been aggregated ($1 - 0/3 = 1$); in the third most of the atomic changes have been aggregated, but not all ($1 - 1/5 = 0.8$).

14.5 Applying metrics

The main use of the 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.

14.5.1 A two-phase process to evaluate algorithms through metrics

The delta model and metrics are independent from a specific data format. Such a level of abstraction, on the one hand, makes it possible to capture relevant peculiarities of changes and to compare heterogeneous input; on the other, it requires a further step for measuring actual delta files. In fact, deltas need to be translated into the universal data model, on top of which metrics can be applied.

There is also another tricky issue. The algorithms use deltas as conceptual objects during their computations while the result of said computation is stored or transmitted in form of serialized files, i.e. patches. These patches almost always contain few information about the changes found and their encapsulation, *even if that information has been detected and exploited by the algorithm.* The point is that the algorithms produce an output that is *optimized* for the application that is expected to process it and to use it for re-building the newer document form 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).

A harder problem is that some information is hidden in such serialized formats. For instance, some changes that the algorithm calculates as aggregates might be finally stored as separate (small) changes and linked each others through identifiers. This is the strategy used by JNDiff [17] and XyDiff [15]: this choice does not match the conceptual model behind the algorithm, rather it is a way to simplify the backward application of the patch.

There is even a worse scenario. Consider for instance Faxma [32]. It 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.

Thus, the process of applying metrics to study algorithms can be refined in two steps:

1. interpretation: the pre-processing analysis in which the algorithm's internal model is made explicit and mapped into the universal one;

2. evaluation: the actual measurement of atomic indicators and aggregated metrics on the pre-processed delta.

While the second step can be generalized and automated (and we actually implemented some tools for this purpose, mentioned in the following section), the first one is different for each algorithm. More important, it requires users to understand the basic functioning of the algorithm.

In fact, there are two approaches to this problem: (1) patching the implementations to store more information in the patches or (2) deducing the missing information from the patches knowing how the serialization process works.

Yet, in both cases these reconstructed deltas are only approximations of the real deltas internally used by the algorithms. They are however the only means that makes the use of these metrics possible, short of rewriting the implementations of the algorithms to produce a dump of the internal representation of the processed deltas. It must also be noted that there are various degree of approximation, as show in figure 14.6: the patches themselves (Φ) can serve as a very crude approximation of the initial delta (Δ), better approximations can be generated with the addition of knowledge about how an algorithm works (the various functions $\tilde{\sigma}^{-1}$, approximate inverse of the serialization function σ).

$$\begin{tikzcd}
\Delta \text{ (delta)} \arrow[rrr, bend left, "\sigma"] & \Delta'' & \Delta' \arrow[l, bend left, "\tilde{\sigma}_2^{-1}"] & \Phi \text{ (patch)} \arrow[l, bend left, "\tilde{\sigma}_1^{-1}"]
\end{tikzcd}$$

Figure 14.6: From delta to patch and back

Finally, note that the preliminary interpretation is to be done once for each algorithm and, from then on, the quality of [the output of] the algorithm can be evaluated in a fully objective manner.

14.5.2 Experimental results on XML diff

In order to test the applicability of the metrics on real deltas we studied three well-known XML diff tools: JNDiff [17], XyDiff [15] and Faxma [32]. We run experiments on the same dataset used to evaluate the 'naturalness' of JNDiff, available at http://twprojects.cs.unibo.it/jndiff-tests/. It consists of five documents, each available in two versions:

- The first two documents are taken from the evaluation of DocTreeDiff [43]: 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 schoolwork project. It is structured in sections and subsections and contains a lot of internal references and code snippets.

We selected this dataset for three main reasons:

1. Input documents are real and taken from heterogeneous sources. Some of them, in turn, were used to evaluate the quality of other diff algorithms. Thus, they cover a wide range of cases. Moreover, they are very different in terms of size, depth, number of elements and attributes, and internal structure.

2. We know the internal structures of the documents and the types of changes applied to them. Thus, we can verify whether or not there is a relation between the types of documents/changes and the behavior of each algorithm.

3. Last but not least, we know the algorithms quite well, from our previous analysis of published papers, of their internal documentation and source code. Thus, we can perform the above-mentioned interpretation of deltas in reasonable time and with reasonable confidence.

		Precision	**Conciseness**	**Meaningfullness**	**Aggregation**	**Economy**
BIBLIO	*JNDiff*	0.23	0.39	0.13	0.42	0.43
	XyDiff	0.09	0.01	0	0.01	0.9
	Faxma	0.09	0.95	0.21	0.96	0
	Trivial	0.07	0	0	0	0.9
LETTER	*JNDiff*	0.38	0	0	0	0.34
	XyDiff	0.28	0	0	0	0.17
	Faxma	0.44	0.58	0.15	0.62	0
	Trivial	0.18	0	0	0	0.84
DL1184	*JNDiff*	0.26	0.54	019	0.59	0.26
	XyDiff	0.05	0	0	0	0.76
	Faxma	0.06	0.69	0.18	0.73	0
	Trivial	0.03	0	0	0	0.88
DL2221	*JNDiff*	0.43	0.64	0.38	0.74	0.28
	XyDiff	0.08	0.28	0.21	0.33	0.32
	Faxma	0.07	0.79	0.26	0.83	0
	Trivial	0.03	0	0	0	0.87
PROTOCOL	*JNDiff*	0.26	0.38	0.14	0.42	0.24
	XyDiff	0.06	0.25	0.07	0.26	0.07
	Faxma	0.06	0.56	0.09	0.58	0
	Trivial	0.06	0	0	0	0.89

Table 14.1: Evaluating metrics for all algorithms on all files of the dataset

According to the previous schema, we first interpreted the results of each algorithm. For JNDiff and XyDiff we constructed augmented patches from the patches generated by the reference implementations. The changes found in the original patches, almost all atomic changes, have been grouped in complex changes similar to those used internally by these algorithms. For Faxma we patched the code available at https://github.com/ept/fuego-diff to produce a low-level dump of the changes detected internally instead of refining the produced patches; the new code is available at https://github.com/gioele/fuego-diff. We could not augment the patches generated by the original code because the serialization format lost too much information as discussed earlier. For instance, Faxma does not store any information about deleted content (the delta only contains references to the original source intermixed with newly added and/or updated content); 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. Once we started extracting the changes directly from the code, we managed to obtain a reasonable approximation that gave us good insights about the behavior of this algorithm too.

The evaluation phase was fully automated. Table 14.1 shows all results we collected. Each column represent 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.

In order 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. This delta is correct and makes it possible to rebuild the newer document from the older one. On the other hand, it provides too little information and is very far from being useful to understand what changed between the two documents. It is anyway interesting to verify if the metrics highlight clearly such 'bad' behavior.

Conciseness, meaningfulness and aggregation could be evaluated in an absolute way by measuring and aggregating atomic indicators with straightforward XPath expressions.

The overall value of precision (including the precision on elements, attributes and texts) has been approximated. The precision indicates the ratio between the modified elements and the touched ones. An exact value could be calculated by knowing exactly what changed and by looking manually at exact modified nodes for each algorithm. A faster but still reliable process consists of considering the number of modified nodes as equal to the minimum amount of modified content between the two input documents. This value can be calculated easily by using external binary diff tools on the isolated content; this value provides a reasonable round-up, since any other algorithm cannot have a higher precision (as it cannot modify a lower number of nodes).

Given that the population value indicates the total number of changes each delta is composed of, absolute values of deltas produced by different algorithms vary a lot and cannot be compared directly. We synthesized a new dimension, called *economy*, to compare the population scores. To solve the problem we first calculated the magnitude of each population, as $P_{algorithm} = \log_{10}\left(Population\left(\delta\right)\right)$ and then calculated:

$$Economy\left(\delta\right) =$$
$$= 1 - \frac{P_{algorithm}}{\max\left(P_{JNDiff},\, P_{XyDiff},\, P_{faxma},\, P_{trivial}\right)}$$

Thus, we obtained a *economy* score between 0 and 1, with higher values for lower (normalized) values of population. This normalization is also the reason why, for each document, there is always an algorithm with economy equal to zero (the one with highest population).

The results of the application of the metrics to documents BIBLIO, PROTOCOL and DL2221 are summarized with a different view in Figure 14.7.

The 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.

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 economy (since only two changes were detected) but obtained a zero score for meaningfulness, conciseness and aggregation since it does not try to give a higher-level interpretation of changes. For the same reason, the precision is the lowest one in all cases. Note that a mere evaluation of the number of edits would have given a very high-quality score to the delta produced by this algorithm.

Consider also the behavior of Faxma wrt aggregation and conciseness. These two dimensions are related each other and capture whether or not an algorithm is able to aggregate changes or only generates mostly atomic change. Both these values are 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 a possible. These two factors lead Faxma to produce deltas with low meaningfullness values. However, this helps Faxma with two of its aims: move changes are translated in “copy” operations in their patch format and reduces the number of changes to be stored in the patch; the ability to produce higher-level interpretation of changes is exchanged for more speed and less space requirements.

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 meaningfulness 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 economy 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 precision score.

The results on JNDiff are also meaningful. The algorithm, in fact, works very well on textual changes and is able to aggregate fine-grained modifications on text nodes into aggregated changes. On the other hand, such aggregation 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 on 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 precision, that is the highest one in all cases. Similarly, the fact that JNDiff tries to reduce the number of detected edits influences the economy score, that is always quite high.

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 14.8.

The plot looks apparently different from the previous ones: the economy score 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 economy 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.

The final plot, related to document LETTER and shown in Figure 14.9, looks again very different from the others. However, it confirms that the metrics are able to capture some peculiarities of each algorithm when dealing with some types of changes. In this case, in fact, the precision of Faxma is very high. The reason is that Faxma is very precise in detecting moves and aggregations of sequences after the deletion of interposing elements. Since the modifications on this document were mainly attributes updates, moves and a few changes on text elements (and no one on mixed content-models) Faxma worked very well. It is not a case that, simultaneously, the results of JNDiff get worse: the capability of detecting changes on mixed content-models and produce a small set of edits is not very helpful in this content. That is the reason why conciseness and aggregation are a bit lower than usual.

For similar reasons - since the impact of nested changes and fine-grained textual modifications is very low - the values of precision and economy for XyDiff are a bit higher than the previous cases.

In conclusion, experiments confirmed that the metrics are able to capture the general behavior of an algorithm, although they only operate on its output. Whereas the output is peculiar, these metrics can also be used as indicators of anomalies or occurrences of specific classes of changes.

We could even envision a document-based comparison of diff results. Users could run different algorithms on the same documents and decide which is the best fit for that specific case. This approach would not be too expensive but would provide users a finer selection process. That is also in line with our initial assumption: the measurement of quality cannot be an absolute and unique value but has to be inflected for each context and for each user.

14.6 Conclusions and future works

This chapter presented a set of objective metrics and a methodology for the evaluation of the qualities of the deltas produced by diff algorithms. These metrics are based on the delta model described in section 14.4. The results in section 14.5.2 show that the values of these metrics reflect real properties of the analyzed algorithms, for example the tendency to detect many localized small changes instead of fewer big changes.

A distinctive point of these 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 changes in a block of text to make the algorithm emit a single big 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 an high conciseness value”.

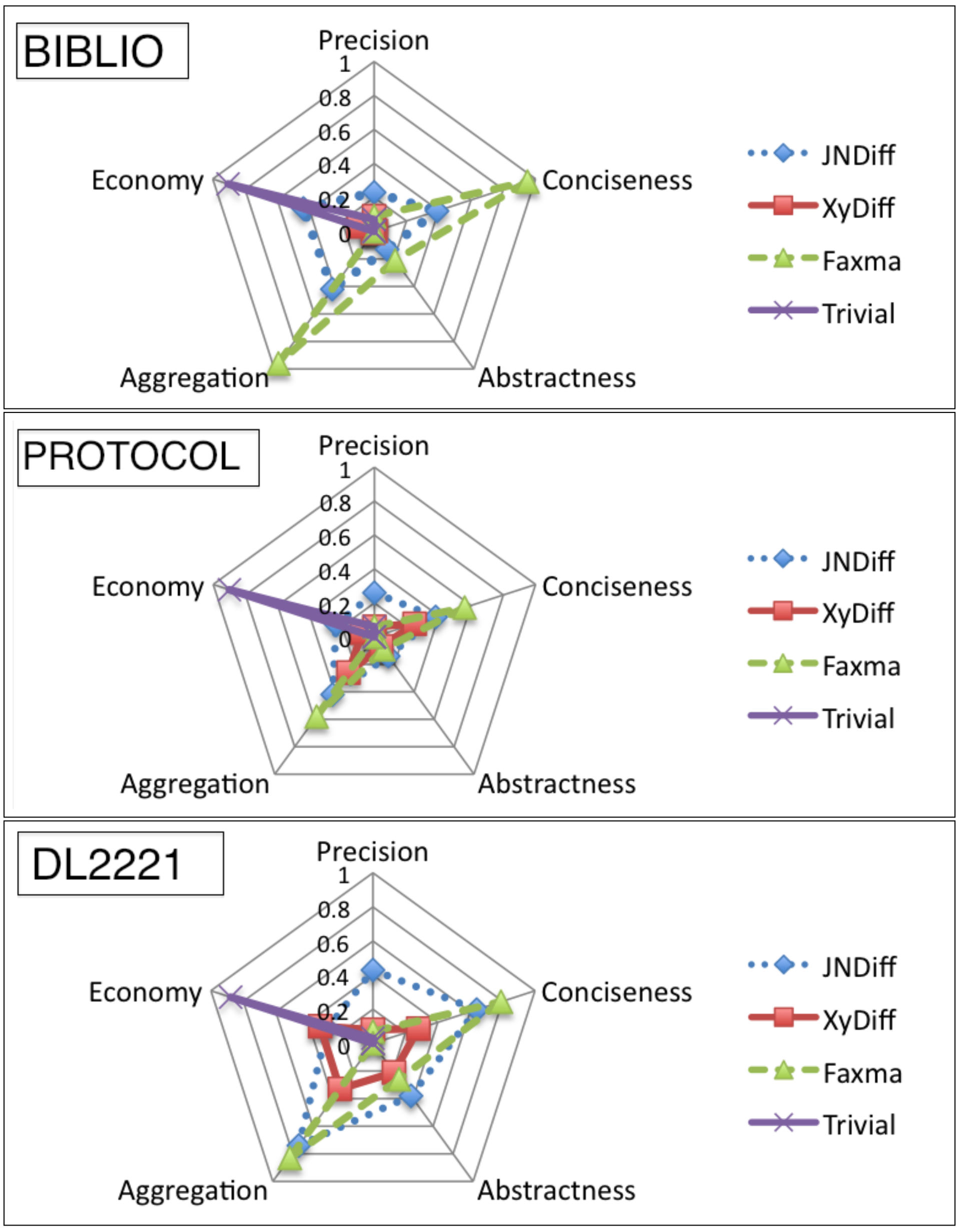

Figure 14.7: Evaluating metrics on documents BIBLIO, PROTOCOL and DL2221

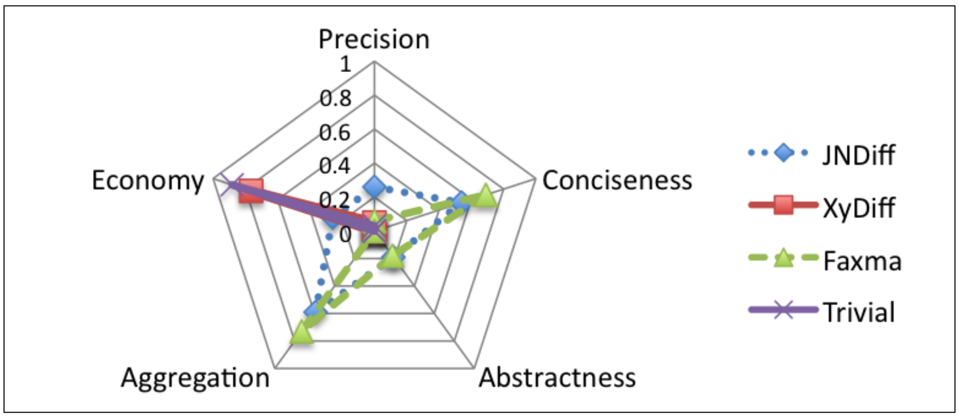

Figure 14.8: Evaluating metrics on document DL1184

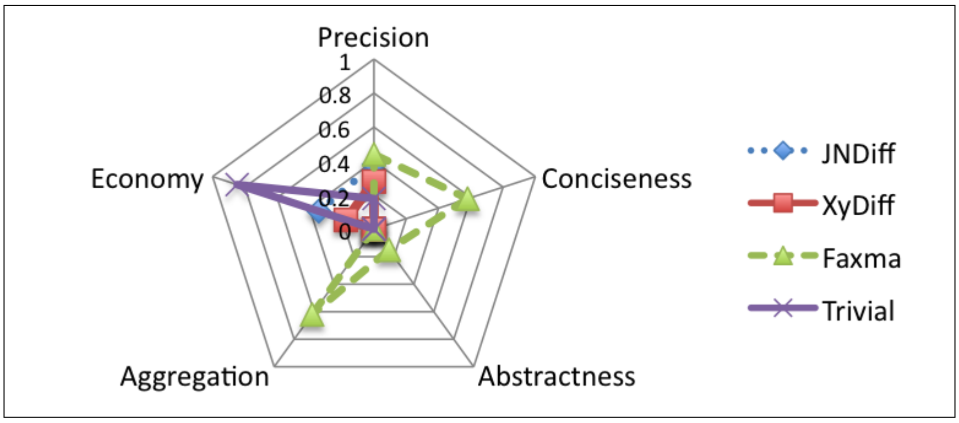

Figure 14.9: Evaluating metrics on document LETTER

Chapter 15

Evolution of ontologies

This chapter describes OntoEv, a tool that analyzes chains of ontology versions and extract trends about how the ontology have been developed and what are the most easily modified entities. The basis of the tool is new algorithm similar to PROMPTDiff [37] and Papavassiliou 2009 [38] but based on the proposed delta model. The OntoEv tool is available as a work in progress at http://barabucc.web.cs.unibo.it/ontoev/.

15.1 Introduction

The reason beyond the creation of OntoEv is the desire to be able to identify and study the various phases of the development that happens during the development of ontologies without having to resort to change summaries or versioning information, things that are rarely available for ontologies found "in the wild", and, anyway, of little use when the this analysis is to be performed on dozens of ontologies.

The development of ontologies can be carried out in many ways. Sometimes ontologies are developed starting from a known domain vocabulary, other times properties are designed first and classes added later as a structure emerges, yet other times it is the opposite: the classes are created first and properties are added as the need arises. Through the analysis of chains of versions of ontologies it is possible to highlight these changes, spotting trends and development patterns.

15.2 The discovery process

OntoEv takes a chain of ontologies snapshots (a linear sequence of snapshots of an ontology in chronological order) and produces a report that shows the phases of development that the ontology has gone through. The analysis performed by

Onto Ev is composed of four steps: first it produces a basic delta with the changes detected between each pair of subsequent versions; then this delta is refined into a more meaningful delta containing complex changes specifically identified for use with the OWL model; later the changes stored in the delta are clustered into broad development activities (e.g. “documentation”, “link to external ontologies”, “growth”, etc.); finally, a visual report is generated, highlighting the flow of the development activities through the chain of versions.

15.2.1 Chains of ontologies

OntoEv operates on chains of ontologies, i.e. linear sequences of different versions of ontology files. An example of ontology chain is the set of revisions of the FOAF ontology :

- http://xmlns.com/foaf/spec/20100809.rdf
- http://xmlns.com/foaf/spec/20100101.rdf
- http://xmlns.com/foaf/spec/20091215.rdf
- etc.

The chains used during the development and testing of OntoEv have been retrieved using the Watson search engine [16] and its module for the discovery of chains based on similarity measures [1].

15.2.2 Creation of deltas

The initial delta, internally referred to as the structural delta, is produced by the OntoVCS [54] library. The deltas produced by OntoVCS, already analyzed in section 3.1.3.2, are atomic changes that only represent additions and deletions of RDF or OWL constructs. An example delta produced by OntoVCS is the list of changes shown in figure 15.1.

```
+ SubClassOf(testbed:Slope testbed:Land_Feature)
- SubClassOf(testbed:Slope testbed:Thing)
+ DataPropertyRange(testbed:is_manmade xsd:boolean)
+ Declaration(DataProperty(testbed:is_manmade))
+ AnnotationAssertion(rdfs:label testbed:is_manmade "is_manmade"@en)
+ DataPropertyDomain(testbed:is_manmade testbed:Natural_Feature)
+ AnnotationAssertion(rdfs:comment testbed:is_manmade
   "indicates whether a feature is manmade or natural.")
```

Figure 15.1: Delta produced by OntoVCS

15.2.3 Refinement of deltas

The initial delta produced using OntoVCS is not rich enough in information to be used for any kind of advanced analysis of the modification made to the ontology. For this reason a new algorithm has been developed to refine this initial delta. The new algorithm aggregates the initial changes into complex meaningful changes that try to match the intent of the user that lead to the detected changes.

The aggregation process of the OntoEv algorithm is a fixed point algorithm similar to PROMPTDiff [37], already discussed in section 3.1.3.1. The algorithm iteratively process all the changes trying to match each candidate with a set of preconditions dictated by the defined aggregation rules. Of all the rules whose preconditions are met by the candidates, the most “complex” is applied. The application of the rule aggregates the matched candidates into a complex change and replaces the matched candidates in the set of candidates with the generated complex change. The convergence of this fixed point algorithm is guaranteed by the monotonicity of the aggregation process: there are no rules that generate more changes than the number of changes they aggregate.

This refinement process augment the delta with meaningful changes that allow the development of simpler analysis. The initial delta show in figure 15.1 is refined by the algorithm into the delta shown in figure 15.2.

```
ClassSpecialization(Slope)
   + ClassSubClassOfAdded(Slope) { subclass of Land_Feature }
   + ClassSubClassOfRemoved(Slope) { subclass of Thing }

DataPropertyOntologyExpansion(is_manmade)
   + DataPropertyDeclarationAdded(is_manmade)
   + DataPropertyConstraintsAddition(is_manmade)
      + DataPropertyDomainAddition(is_manmade)
         + DataPropertyDomainAdded(is_manmade) {
            domain: Natural_Feature }
      + DataPropertyRangeAddition(is_manmade)
         + DataPropertyRangeAdded(is_manmade) {
            range: boolean }
   + ClassDocumentationAdded(is_manmade) {
      comment: "indicates whether a ..." }
   + ClassDocumentationAdded(is_manmade) {
      label: "is_manmade"@en }
```

Figure 15.2: Delta refined by OntoEv

15.2.4 Clustering of changes

Once a delta with meaningful changes has been generated by the OntoEv algorithm these changes are clustered and classified. For instance, changes such as DataPropertyConstraintsAddition and ClassSpecialization are classified as *ontology refactoring* while changes such as DataPropertyOntologyExpansion and ClassAddition are classified as *ontology expansion*.

15.3 Results from the analysis of a chain

The reports generated by OntoEv analyzing the set deltas calculated on the chain highlights various aspects of what has changed: the various phases that can be seen in the development of the ontology, the complexity of the changes that have been made and what are the entities that have received the most attention during the development of the ontology.

The change in development focus is shown, as depicted in figure 15.3, by

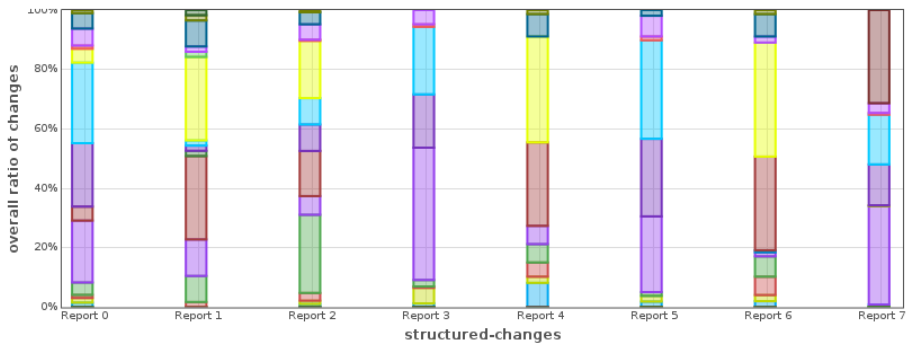

Figure 15.3: Example of development flow

The complexity metric shows how much an ontology has changed during a certain revision. For example the ontology shown in figure 15.4 has been developed mostly through simple changes, a fact that is highlighted by the high number of changes of complexity 2 in all the 8 revisions of the ontology. The complexity graph shows at the same time how much an ontology has changed (the height of the stacked bars) and how convoluted are the changes (the distribution of the complexity on the horizontal axis).

Finally, the OntoEv tools reports what are the entities that received most attention. There are various actions that OntoEv takes into account to calculate what are the entities that have been modified the most: first it calculated the number of their properties have been changed, the the number of refactoring changes that

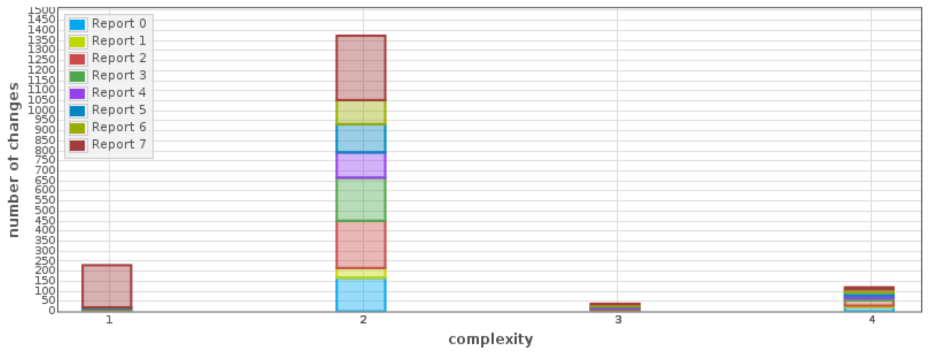

Figure 15.4: Example of complexity report/graph

happened to them and, finally, the number of times they have been referred to in changes made to other entities.

Chapter 16

Conclusions

The goals of this thesis are to formalize what it means to find differences between documents and to find a single shared formalization that can be used by any algorithm to describe the differences found between any kind of comparable documents. The main contribution of this thesis is a universal model of deltas and documents. This model meets the given goals as it is able to express deltas produced by different algorithms that operate on different kinds of documents and that are based on arbitrary sets of recognized operations.

The goals of this thesis must be fulfilled for both theoretical and practical reasons. From a theoretical point of view, without a shared formalization of diff algorithms and deltas it is difficult to compare different algorithms, their abilities and their performances. From a practical point of view, the lack a single formalization and reference API makes it hard or impossible to create tools that support more than one diff algorithm, a need that is often need, for example, for XML-based formats like DocBook where one may be interested in seeing the differences both at the XML level and at the DocBook level.

The starting point of this research has been the analysis of the way existing algorithms operate and how they store the information about the differences they have found. In addition to the models embedded in the algorithms, also standalone models have been studied and compared. The analysis of all these models showed that they all perform similar operations and operate on similar data structures; this means that it is possible to unify all these models under a single model.

The main scientific contribution of this thesis is a universal delta model that can be used to represent the changes found by an algorithm. The main part of this model are the formal definition of changes (the pieces of information that records that something has changed), operations (the definitions of the kind of change that happened) and deltas (coherent summaries of what has changed between two documents). The fundamental mechanism that makes the changes as defined in the universal delta model a very expressive tool, is the use of encapsulation relations between changes. In the universal delta model, changes are not simple records of what has changed, they can also be combined into more complex changes that express the fact that the algorithm has detected a more nuanced kind of change. For example, the change represent the addition of a chapter heading and the changes representing the the addition of the several single paragraphs can all be encapsulated into a more meaningful change that describe the addition of a chapter.

In addition to the main entities (i.e., changes, operations and deltas), the model describes and defines also documents and the concept of equivalence between documents. As a corollary to the model, there is also an extensible catalog of possible operations that algorithms can detect, used to create a common library of operations, and an UML serialization of the model, useful as a reference when implementing APIs that deal with deltas.

The model has been successfully used in two experiments. In the first experiment, described in chapter 14, the model is exploited to compare the qualities of deltas generated by various algorithms. Among these qualities one can find the precision of a delta (how much non-needed information is contained in the delta) or its conciseness (how many changes are used to describe the differences between two documents). The analysis of the qualities is based on data extracted in an objective way from the deltas, deltas that have been modeled using the universal delta model. This methodology removes much of the subjectivity from the analysis of deltas, making it possible to create automatic tools that perform quality analysis on the deltas without human supervision. In the second experiment, instead, the universal delta model is used to express the differences between various snapshots of an OWL ontology. The produced deltas are used to detect the various phases of development the ontology has gone through, for example when documentation has been added to it or when the main focus of development has been the integration of other external ontologies.

The two experiments highlighted that the use or the conversion of deltas to the universal delta model does not reduce the expressivity of the existing deltas. On the contrary, the model allows the creation of more meaningful deltas and makes it easier to write tools that perform significant analysis on the deltas.

The universal delta model presented in this thesis acts as the formal groundwork upon which algorithm can be based and libraries can be implemented. It removes the need to recreate a new delta model and terminology whenever a new algorithm is devised. It also alleviates the problems that toolmakers have when adapting their software to new diff algorithms. The universal delta model forms a solid foundation for future research in the field of difference algorithms.

Bibliography

[1] Carlo Allocca. Automatic identification of ontology versions using machine learning techniques. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*, volume 6643 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2011.

[2] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pages 2–13. IEEE Computer Society, 2004.

[3] Gioele Barabucci, Luca Cervone, Angelo Di Iorio, Monica Palmirani, Silvio Peroni, and Fabio Vitali. Managing semantics in XML vocabularies: an experience in the legal and legislative domain. In *Proceedings of Balisage: The Markup Conference 2010*, volume 5 of *Balisage Series on Markup Technologies*, 2010.

[4] Gioele Barabucci, Luca Cervone, Monica Palmirani, Silvio Peroni, and Fabio Vitali. Multi-layer markup and ontological structures in Akoma Ntoso. In Pompeu Casanovas, Ugo Pagallo, Giovanni Sartor, and Gianmaria Ajani, editors, *AI Approaches to the Complexity of Legal Systems. Complex Systems, the Semantic Web, Ontologies, Argumentation, and Dialogue - International Workshops AICOL-I/IVR-XXIV Beijing, China, September 19, 2009 and AICOL-II/JURIX 2009, Rotterdam,The Netherlands, December 16, 2009 Revised Selected Papers*, volume 6237 of *Lecture Notes in Computer Science*, pages 133–149. Springer, 2010.

[5] Gioele Barabucci, Paolo Ciancarini, Angelo Di Iorio, and Fabio Vitali. Measuring quality of diff algorithms: quantitative metrics. Submitted for publication in IEEE Transactions on Knowledge and Data Engineering.

[6] Gioele Barabucci, Monica Palmirani, Fabio Vitali, and Luca Cervone. Long-term preservation of legal resources. In Kim Normann Andersen, Enrico Francesconi, Åke Grönlund, and Tom M. van Engers, editors, *Electronic Government and the Information Systems Perspective - Second International Conference, EGOVIS 2011, Toulouse, France, August 29 - September 2, 2011. Proceedings*, volume 6866 of *Lecture Notes in Computer Science*, pages 78–93. Springer, 2011.

[7] Gioele Barabucci, Silvio Peroni, Francesco Poggi, and Fabio Vitali. Embedding semantic annotations within texts: the FRETTA approach. In Sascha Ossowski and Paola Lecca, editors, *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, pages 658–663. ACM, 2012.

[8] Gioele Barabucci and Fabio Vitali. XDTD as a simple validation language for XML-based legal documents. In Guido Governatori, editor, *Legal Knowledge and Information Systems - JURIX 2009: The Twenty-Second Annual Conference on Legal Knowledge and Information Systems, Rotterdam, The Netherlands, 16-18 December 2009*, volume 205 of *Frontiers in Artificial Intelligence and Applications*, pages 1–10. IOS Press, 2009.

[9] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[10] Ronald J. Brachman and Hector J. Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann, 2004.

[11] Randal Burns, Al C. Burns, and Darrell D. E. Long. A linear time, constant space differencing algorithm. In *Performance, Computing, and Communications Conference, 1997*, pages 429–436. IEEE International, feb 1997.

[12] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James Rice. Okbc: A programmatic foundation for knowledge base interoperability. In Jack Mostow and Chuck Rich, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, pages 600–607. AAAI Press / The MIT Press, 1998.

[13] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996*

ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996, pages 493–504. ACM, 1996.

[14] S. V. Coox. Axiomatization of the evolution of XML database schema. *Programming and Computer Software*, 29(3):140–146, 2003.

[15] Grégory Cóbena, Serge Abiteboul, and Amélie Marian. Detecting changes in XML documents. In Rakesh Agrawal and Klaus R. Dittrich, editors, *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 41–52. IEEE Computer Society, 2002.

[16] Mathieu d'Aquin and Enrico Motta. Watson, more than a semantic web search engine. *Semantic Web*, 2(1):55–63, 2011.

[17] Angelo Di Iorio, Michele Schirinzi, Fabio Vitali, and Carlo Marchetti. A natural and multi-layered approach to detect changes in tree-based textual documents. In Joaquim Filipe and José Cordeiro, editors, *Enterprise Information Systems, 11th International Conference, ICEIS 2009, Milan, Italy, May 6-10, 2009. Proceedings*, volume 24 of *Lecture Notes in Business Information Processing*, pages 90–101. Springer, 2009.

[18] Adam Duley, Chris Spandikow, and Miryung Kim. Vdiff: a program differencing algorithm for verilog hardware description language. *Autom. Softw. Eng.*, 19(4):459–490, 2012.

[19] Mohamed El-Attar. UseCaseDiff: an algorithm for differencing use case models. In *9th International Conference on Software Engineering Research, Management and Applications, SERA 2011, Baltimore, MD, USA, August 10-12, 2011*, pages 148–152. IEEE Computer Society, 2011.

[20] Rafael S. Gonçalves, Bijan Parsia, and Ulrike Sattler. Ecco: a hybrid diff tool for OWL 2 ontologies. In Pavel Klinov and Matthew Horridge, editors, *Proceedings of OWL: Experiences and Directions Workshop 2012, Heraklion, Crete, Greece, May 27-28, 2012*, volume 849 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[21] Michael Hartung, Toralf Kirsten, Anika Gross, and Erhard Rahm. Onex: Exploring changes in life science ontologies. *BMC Bioinformatics*, 10, 2009.

[22] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In Bernard N. Fischer, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 234–245. ACM, 1990.

[23] J. W. Hunt and M.D. McIllroy. An algorithm for differential file comparison. Technical Report 41, AT&T Bell Laboratories Inc., 1976.

[24] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[25] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic.* IEEE, New York, NY, USA, August 2008.

[26] Daniel Jackson and David A. Ladd. Semantic diff: a tool for summarizing the effects of modifications. In Hausi A. Müller and Mari Georges, editors, *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*, pages 243–252. IEEE Computer Society, 1994.

[27] Jaakko Kangasharju and Tancred Lindholm. A sequence-based type-aware interface for XML processing. In M. H. Hamza, editor, *Internet and Multimedia Systems and Applications, EuroIMSA 2005, Grindelwald, Switzerland, February 21-23, 2005*, pages 83–88. IASTED/ACTA Press, 2005.

[28] Michel Klein. *Change Management for Distributed Ontologies.* PhD thesis, Vrije Universiteit Amsterdam, August 2004.

[29] Boris Konev, Carsten Lutz, Dirk Walther, and Frank Wolter. Logical difference and module extraction with cex and mex. In Franz Baader, Carsten Lutz, and Boris Motik, editors, *Proceedings of the 21st International Workshop on Description Logics (DL2008), Dresden, Germany, May 13-16, 2008*, volume 353 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[30] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 282 –290, nov 1992.

[31] Erwin Leonardi and Sourav S. Bhowmick. Xandy: A scalable change detection technique for ordered XML documents using relational databases. *Data & Knowledge Engineering*, 59(2):476–507, 2006.

[32] Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. Fast and simple XML tree differencing by sequence alignment. In Dick C. A. Bulterman and David F. Brailsford, editors, *Proceedings of the 2006 ACM Symposium on Document Engineering, Amsterdam, The Netherlands, October 10-13, 2006*, pages 75–84. ACM, 2006.

[33] Webb Miller and Eugene W. Myers. A file comparison program. *Softw., Pract. Exper.*, 15(11):1025–1040, 1985.

[34] Gilles Muller, Yoann Padioleau, Julia L. Lawall, and René Rydhof Hansen. Semantic patches considered helpful. *Operating Systems Review*, 40(3):90–92, 2006.

[35] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[36] Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982.

[37] Natalya Fridman Noy and Mark A. Musen. PROMPTDIFF: a fixed-point algorithm for comparing ontology versions. In Rina Dechter and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 744–750. AAAI Press / The MIT Press, 2002.

[38] Vicky Papavassiliou, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, and Vassilis Christophides. On detecting high-level changes in RDF/S KBs. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, volume 5823 of *Lecture Notes in Computer Science*, pages 473–488. Springer, 2009.

[39] Peter Plessers, Olga De Troyer, and Sven Casteleyn. Understanding ontology evolution: A change detection approach. *Journal of Web Semantics*, 5(1):39–49, 2007.

[40] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, pages 188–197. IEEE Computer Society, 2004.

[41] Sebastian Rönnau and Uwe M. Borghoff. Versioning XML-based office documents. *Multimedia Tools and Applications*, 43(3):253–274, 2009.

[42] Sebastian Rönnau, Christian Pauli, and Uwe M. Borghoff. Merging changes in XML documents using reliable context fingerprints. In Maria da Graça Campos Pimentel, Dick C. A. Bulterman, and Luiz Fernando Gomes Soares, editors, *Proceedings of the 2008 ACM Symposium on Document Engineering, Sao Paulo, Brazil, September 16-19, 2008*, pages 52–61, 2008.

[43] Sebastian Rönnau, Geraint Philipp, and Uwe M. Borghoff. Efficient change control of XML documents. In Uwe M. Borghoff and Boris Chidlovskii, editors, *Proceedings of the 2009 ACM Symposium on Document Engineering, Munich, Germany, September 16-18, 2009*, pages 3–12. ACM, 2009.

[44] Jason Stanek, Suraj Kothari, and Kang Gui. Method of comparing graph differencing algorithms for software differencing. In *2008 IEEE International Conference on Electro/Information Technology, EIT 2008, held at Iowa State University, Ames, Iowa, USA, May 18-20, 2008*, pages 482–487. IEEE Computer Society, 2008.

[45] Sathya Sundaram and Sanjay K. Madria. A change detection system for unordered XML data using a relational model. *Data & Knowledge Engineering*, 72:257–284, 2012.

[46] Jean-Yves Vion-Dury. Diffing, patching and merging XML documents: toward a generic calculus of editing deltas. In Apostolos Antonacopoulos, Michael J. Gormish, and Rolf Ingold, editors, *Proceedings of the 2010 ACM Symposium on Document Engineering, Manchester, United Kingdom, September 21-24, 2010*, pages 191–194. ACM, 2010.

[47] Jean-Yves Vion-Dury. A generic calculus of XML editing deltas. In Matthew R. B. Hardy and Frank Wm. Tompa, editors, *Proceedings of the 2011 ACM Symposium on Document Engineering, Mountain View, CA, USA, September 19-22, 2011*, pages 113–120. ACM, 2011.

[48] Yuan Wang, David J. DeWitt, and Jin yi Cai. X-Diff: an effective change detection algorithm for XML documents. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 519–530. IEEE Computer Society, 2003.

[49] Richard Saul Wurman. *Information architects.* Graphis Inc, 1997.

[50] Dimitris Zeginis, Yannis Tzitzikas, and Vassilis Christophides. On the foundations of computing deltas between RDF models. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 637–651. Springer, 2007.

[51] Dimitris Zeginis, Yannis Tzitzikas, and Vassilis Christophides. On computing deltas of RDF/S knowledge bases. *ACM Transactions on the Web*, 5(3):14, 2011.

[52] Kaizhong Zhang, Jason Tsong-Li Wang, and Dennis Shasha. On the editing distance between undirected acyclic graphs and related problems. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, 1995.

[53] Xiangyu Zhang and Rajiv Gupta. Matching execution histories of program versions. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 197–206. ACM, 2005.

Sitography

[54] OntoVCS. http://code.google.com/p/ontovcs/. accessed Nov 26, 2012.

[55] Tim Berners-Lee and Dan Connolly. Delta: an ontology for the distribution of differences between RDF graphs. http://www.w3.org/DesignIssues/Diff, 2009. accessed Nov 26, 2012.

[56] John Boyer and Glenn Marcy. Canonical XML Version 1.1. Recommendation, W3C, May 2008. http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/. Latest version available at http://www.w3.org/TR/xml-c14n11.

[57] Bram Cohen. Patience diff advantages. http://bramcohen.livejournal.com/73318.html, 2010. accessed Nov 26, 2012.

[58] Mark-Jason Dominus. Algorithm::diff: Longest common subsequence algorithm. http://perl.plover.com/diff/Diff.pm, 1999. accessed Nov 26, 2012.

[59] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. Recommendation, W3C, November 2009. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/. Latest version available at http://www.w3.org/TR/owl2-overview/.

[60] Ramanathan V. Guha and Dan Brickley. RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/. Latest version available at http://www.w3.org/TR/rdf-schema.

[61] Arnaud Le Hors, David Raggett, and Ian Jacobs. HTML 4.01 Specification. Technical report, 1999. http://www.w3.org/TR/1999/REC-html401-19991224. Latest version available at http://www.w3.org/TR/html401.

[62] Ned Konz. Algorithm::diff: Compute intelligent differences between two files / lists. http://search.cpan.org/~nedkonz/Algorithm-Diff/lib/Algorithm/Diff.pm, 2002. accessed Nov 26, 2012.

[63] Peter F. Patel-Schneider, Boris Motik, and Bernardo Cuenca Grau. OWL 2 Web Ontology Language Direct Semantics. Recommendation, W3C, November 2009. http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/. Latest version available at http://www.w3.org/TR/owl2-direct-semantics/.

[64] Colin Percival. Naive differences of executable code. http://www.daemonology.net/bsdiff/, accessed Nov 26, 2012, 2003.

[65] SyncRO Soft SRL. oXygen XML diff & merge. http://www.oxygenxml.com/xml_editor/xml_diff_and_merge.html, 2012. accessed Nov 26, 2012.

[66] The GNU Diffutils authors. *Comparing and Merging Files*, chapter Detailed Description of Context Format. 2012. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Context.html, accessed Nov 26, 2012.

[67] The GNU Diffutils authors. *Comparing and Merging Files*, chapter Detailed Description of Unified Format. 2012. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html, accessed Nov 26, 2012.

[68] Norman Walsh. The DocBook schema version 5.0. Technical report, OASIS, 2008. http://docbook.org/specs/docbook-5.0-spec-cd-03.html.