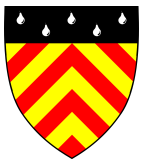
UNIVERSITY OF
CAMBRIDGE

Electronic Long-Term Archiving of Complex Textual Artefacts

Daniel Bruder



Clare Hall

This dissertation is submitted on May 4, 2023 for the degree of Doctor of Philosophy

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee

Daniel Bruder
May 4, 2023

Abstract

Daniel Bruder – Electronic Long-Term Archiving of Complex Textual Artefacts

Digital long-term archiving and data curation, whether in the Digital Humanities (DH) or elsewhere, depends on a suitable data model and must fulfill many requirements. One prerequisite for long-term archiving is interoperability of the data, across machines, computer architectures, and operating systems. But there are many use cases in philology where the requirements are even more stringent, for example the philological reconstruction of textual artefacts and their gestation. Such reconstruction depends on a format which natively supports non-linear text. It should also provide native support for multiple hierarchies over the data. In addition, the format should ideally enable different teams of philologists to work together successfully and sustainably on the same project, and over long stretches of time. Documents in this format should therefore be easily readable for humans, while still being machine-readable. In this work, I show that the two document models in common use today fall short of these requirements.

I then set out to provide my solution to the problem: a topological document format in which symbols gain their meaning through their topological arrangement. Annotation is expressed in a stand-off manner and therefore able to support multiple hierarchies and concurrent text. My design includes operations that can programmatically support the format: how to create data in the format, how to access and mutate the data by systematic means, how to check whether the data is consistent, and how to print out the data after work in the DH project is concluded, possibly keeping the data in that format for centuries. One part of the solution is to extend the classic `diff` model of editorial operations by adding an open variant. Structural data access and mutation in my document model relies on Region Algebra, which was invented in 2002 by Miller. String search over the non-linear data uses an object called a variant graph, which can be systematically derived from my topological notation. After showing that my solution does not share the problems of its predecessors, I will lay out the implementation of the model. I will also show how import from existing formats works, by using the Wiener Ausgabe as my showcase. The design is based on a combination of insights from philology with techniques from computer science, hopefully enabling philologists to systematicise the editorial operations they use, while exposing computer scientists to interaction techniques from a century-old endeavour.

Contents

Chapter 1

Introduction

1.1 Longevity

Humanity's intellectual achievements are one part of our cultural heritage. Humans of all ages and cultures have shared the intuition that ideas are valuable cultural goods and that it is essential that they be protected and kept for future generations to enjoy and derive knowledge from. That is the only explanation why so many past ideas have managed to survive for so long, in the face of wars, natural disasters, intellectual rivalry, and many other destructive forces that might have wiped them off the face of the earth.

Intellectual achievements might consist of writings by important philosophers, music pieces by composers, novels, and many other cultural artefacts. Such artefacts are intangible, unlike the kind of cultural heritage that consists in physical objects such as buildings, statues, two-dimensional art such as paintings, religious artefacts, engineering solutions and many others. These have also been protected across centuries and millennia, often involving immense cost or even risk to life. Physical cultural artefacts recognised as such are typically maintained in the most protective environment available to the current custodian of the objects. Depending on the object, its size, material and location, they are protected from human touch, humidity, light and other destructive forces. During disruptive episodes such as wars or natural catastrophes, valuable cultural objects are generally moved to a safe place, whether this type of rescue was organised by society or done spontaneously.

Intangible goods have been treated with similar care. Ideas need carriers such as clay tablets, paper, or hard drives to store them. The physical stability of the medium is a necessary precondition for an intellectual engagement with the ideas contained in it. The carriers can be treated just as if they themselves were tangible cultural objects, which

covers the *conservation* aspect of long-term archiving. Conservation concerns a specific medium and its material qualities of keeping data. In this thesis, I will focus on a different aspect of long-term archiving, namely *preservation*. Preservation is concerned with the question of how to best structure data in order to retain and keep the actual information contained in it safe and accessible over long stretches of time.

Because ideas are not tied to one particular medium, they can be copied so that many replicas can be made and kept in different places, reducing the risk of destruction. Only when a major accident such as the burning down of the library of Alexandria happens, do ideas actually get lost. But thankfully, such idea extinction events happen rarely.

Paper and ink have served the purpose of preservation of ideas well, but with the invention of the computer, different storage mediums have become available. These have advantages, such as requiring far less physical space, being searchable, allowing to preserve secondary ideas on top of the original ideas, and keeping easier track of who is citing who. However, the electronic storage of ideas and cultural assets also has disadvantages that could endanger the longevity of the ideas. Some of these disadvantages are obvious in that the storage medium itself might have a short life: CDs, whose average material life is 15 years, or hard drives, whose life might be even shorter. A disadvantage that is related to this is the shelf life of the medium as such: if new inventions bring better storage, human society moves beyond a particular medium, and all data needs to be copied over from the old storage method (say, CDs) to the new storage method (say, mp3).

But even if these more superficial problems are solvable, there are more insidious enemies to longevity, as I will explain in detail in chapter 2. If the format of recording primary and secondary information is of the wrong kind, it might offer systematic obstacles to researchers' access to the data. Much of the information we will need to store contains overlapping structures, non-contiguous structures, multiple document hierarchies and even structural changes to documents, posing great obstacles to data access if the wrong format is chosen. Workarounds can be created for most of these problems, but the accumulation of different workarounds in the same structure often creates unnecessary complexity. Some of these obstacles can be observed even today, only years after the general adoption of some format, but they will grow larger with time. This was presaged in 2014 by one of the most important figures in the Digital Humanities domain, Patrick Sahle (translation from German mine):

> *Last but not least, the ineradicable initial assumption that these would be simple problems to be treated with standard solutions has led to the many failures of humanities–computer science cooperations in the last 30 years. At the same time, this basic experience is one of the causes for the development of Digital Humanities. In their historical agnosticism, some of the computer science experts are now repeating*

> *history, while others have recognized that the problems in the humanities are often not trivial, but on the contrary can give valuable impulses for developments in applied as well as in theoretical computer science itself.* Sahle (2014)

In a hundred years, it's very unlikely that any humans living then would be able to extract the information contained in an electronic format unless the format itself is self-explanatory, even if we assume that the purely technical problems of the medium could be solved.

A century is not even a very long time span for a task as big as the one we are facing here. It is even thinkable that there might be a total breakdown of civilisation on this planet, for some unspecified time. In times of such crisis, it might be useful to have the ability to 'print out' the current state of affairs to an emergency format, which can be stored in a safe place. Ideally, the emergency format should record everything that has been thought about the idea up to this point in time, in a systematic format, and it should do so in a self-explanatory way. The future heirs of human culture might then have a running chance of recovering our intangible heritage, in whichever physical format they find most useful, and eventually carry on the human tradition of intellectual interaction.

1.2 Interoperability

Up to now, I have described only one aspect of what we want to do with intangible cultural assets: preserve them for posteriority. But we want to do much more with them. They are ideas, and human minds forever want to interact with ideas. The scholarly discipline that interacts with textual ideas is called philology. For millennia, philologists have commented on primary ideas, combined them, connected ideas with outside events and information, stated hypotheses and tried to verify them using original texts. This activity has led to valuable secondary ideas, often in the form of annotations, written down in some form for future philologists and other readers.

Digital humanities defines itself as the scientific discipline that enables intellectual interchange with existing ideas, coordinating the creation and annotation of new ideas that are connected to the ideas in the text. That task goes far beyond the simpler task of simply preserving primary ideas; it also needs to enable human minds to interact with the ideas and to add to them. Ideally, the interaction should be done in such a way that the storage mechanism is invisible to the philologist. The philologist's daily job should be supported so that what they do comes naturally to them: search for particular instantiations of an idea, add analyses in the form of annotations, save their day's work, exchange their secondary and tertiary ideas with colleagues, change their mind, and add to the body of knowledge in this way.

However, many of today's Digital Humanities projects spend too much time solving problems with the format that records the ideas and the interaction with them. The core of the problem is an over-reliance on the tree model, and the various workarounds that were necessitated by the inadequacies of the tree model when applied to complex manuscripts. We can tell that the storage mechanism actively stands in the way of daily work if access to the ideas for philologists in a project requires a computer science specialist on constant duty. This computer scientist might for example write access programs in XSLT for each different research question the philologist might have. We can also tell that something has gone wrong if the original text cannot be easily separated from the annotations, or if even this basic task requires some form of programming.

What would a truly interoperable way of accessing the data look like? The electronic environment would be such that philologists working on the same text could exchange the results of their labour easily with each other, across continents and across time. It should be possible to search in the text, be it primary, secondary and any subsequent level's, add annotations, and even decide that one wants to add an entire new type of annotation. Such actions are the day-to-day job of philologists, and they shouldn't need stand-by computer scientists to do their job. The computer scientists should design a good data model, a syntax and semantics for it, hand over the system, and then leave the philologists to do their job.

1.3 An example manuscript

An central task in philology is the reconstruction of a manuscript's gestation. To understand what is involved in this task, let's first take a look at a manuscript facsimile of a text piece written by the philosopher Ludwig Wittgenstein. Wittgenstein published only one single book in his lifetime – but left more than 20.000 pages of manuscripts, across a wide range of topics reaching from mathematics, to logic, and language theory, which are jointly called Wittgenstein's Nachlass (english: literary estate)[1]. The importance of preserving Wittgenstein's unique writing is reflected in the fact that the Nachlass was included in the UNESCO "Memory of the World" Program.

Figure 1.1 gives an example of a handwriting-facsimile. We can immediately see that the author revised the text in many ways. Philologists reconstruct such handwritings in minute detail, in order to reconstruct the development and gestation of ideas.

[1]The long history and complexity of encoding Wittgenstein's Nachlass as a textual corpus can be estimated from a (select) list of publications: Pichler 1995; Bazzocchi 2015; Erbacher 2015; Hrachovec 2006; Erbacher 2011; Hadersbeck et al. 2014; Wittgenstein and Nedo 1993–2023; Pichler 2002; Pichler 2007; Stern 2010; Rothhaupt 2008; Hrachovec 2000; Hrachovec 2005.

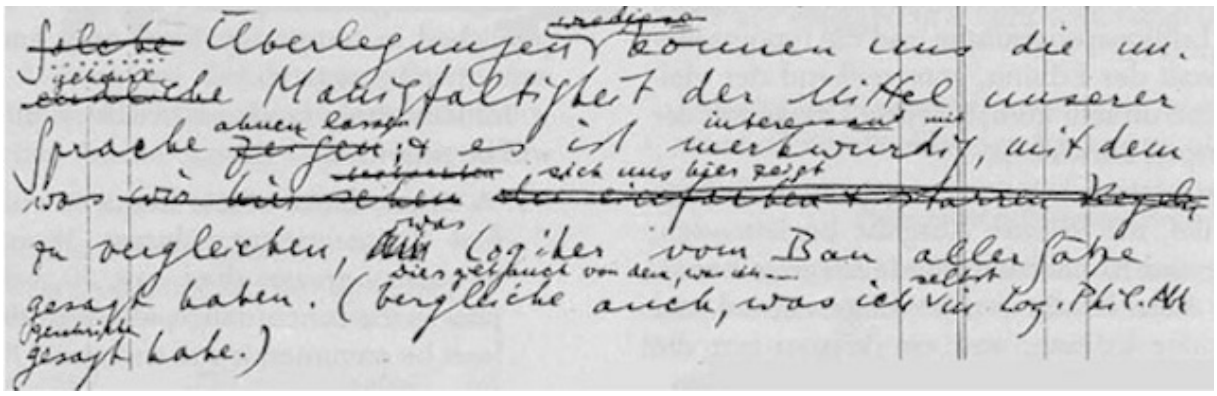


Figure 1.1: Facsimile from the Wittgenstein Nachlass

What follows is the plain-text transcription of the first sentence of the facsimile, translated to English:

> *[Such] considerations /like these/ can {show us/let us fathom} the {infinite/vast} multitude of the means of our language; and it is {odd/interesting} to compare with what we {see/observe/what is revealed to us here}, the simplistic and rigid rules which logicians have proclaimed about the structures of all sentences.*

The transcription above uses diacritical marks within the text to express some of the author's editorial processes, namely:

- */pipe symbols/* to indicate an insertion,
- *[square brackets]* for a deletion, as well as
- *{curly brackets/curly braces}* for undecided, alternate reading variants.

The diacritic marks are used to represent the *editorial operations* (e.g. insertions, deletions, substitutions) within the text.

We can evidently read the text in multiple ways. A *reading* is the text resulting from one of the possible combinations of revisions of the text. Which one of these readings then is the intended interpretation? From an *ex post* position, i.e., after the act of writing has long passed, it is impossible to define a single, authoritative interpretation that is absolutely true. However, out of a plethora of possible readings, what can be done is identify the limited number of readings that appear plausible.

For instance, at the very beginning of the first sentence one can find the deletion of *Such (Solche)* as well as the insertion of *like these (wie diese)*. Since the semantic function of these two expressions is very similar, these two editorial operations can be seen as *coordinated*: they belong together in the sense that one replaces the other. Therefore, the two editorial operations would contribute to the potential readings of this sentence only if they are performed together.

When modelling the reconstruction in this way, one interprets the author's intention: we assume Wittgenstein intended to change the reading from *Such considerations* to *Considerations like these* – as opposed to a tautological reading of *Such considerations like these*. By making this assumption, we have performed an act of *edition*[2] of the text, because our interpretation of the meaning came into play. Edition is opposed to a mere *transcription* of the text, which only describes observable textual phenomena.

Note that editorial operations can be undone (or rather, applied to each other recursively). After replacing *Such* with *like these*, Wittgenstein apparently changed his mind and reverted the cancellation. This is indicated by the dots below *Such*. Presumably simultaneously, he also cancelled its coordinated counterpart *like these*, in effect changing this part of the sentence back to its original state.

Another example of a coordinated editorial option in our text occurs in the change from

> *show us … the infinite multitude of the means of our language*

to the variant reading of

> *let us fathom … the vast multitude of the means of our language*

Again, one can assume these changes were meant to be coordinated, despite their distance in the text, this time because of the syntactic parallelity of the text pieces. A philologist might use the assumption of coordinated editorial options to support, for instance, the hypothesis that the text reflects a mental shift in Wittgenstein's mind, from understanding language as a mathematically precise concept (*show us … infinite multitude*) to an intuitive, approximate understanding (*let us fathom … the vast multitude*).

This example exposes an important phenomenon in Wittgenstein's writing: the *open variant*, which is essentially two or more alternative readings which are left purposefully undecided. Another example of a variant reading can be found in the passage of *what we {see/observe/what is revealed to us here}*. Like insertions and deletions, open variants – as a specific form of an editorial operation –, contribute to different reading possibilities.

Why do open variants occur in anybody's writing at all? In some cases they might happen accidentally, for instance when the author wishes to leave open a final decision on the exact wording and then forgets to make a final choice. In the case of Wittgenstein, experts are reasonably certain that open variants are intentional and crucial to his thinking and writing. Therefore, open variants should not be seen as Wittgenstein being undecided but rather, as a pragmatic means to reach for a very specific meaning. The meaning that he wants to catch seems to often lie between two or more linguistic expressions, where neither one alone would suffice to catch his intended meaning.

[2]In philology, an *edition* of a text is an informed, but nevertheless subjective interpretation of a text.

Considering all these editions and readings, we can see that the reconstruction of the text's development results in a text that can be read in multiple ways and which, by its nature, is *non-linear*. However, many applications in Digital Humanities force us to make non-linear text linear by spelling out the readings. For instance, if we want to search for certain constructions in a text, we need to perform this search on logically linearised text. The operation of spelling out readings is called *linearisation*.

The use of diacritical marks embedded within the text's reconstruction is only one way how non-linear texts and their potential readings could be represented. However, as more and more phenomena of the original text are reconstructed in this way, the more convoluted the transcribed text will become.

Additionally, using such a mechanism in an editorial context, it is near-impossible to record coordinations between distant parts of the text as we have just seen. Alternatively, we could also choose a schematic representation, like the one in listing 1.1.

Listing 1.1 Structural facsimile transcription

```
Solche              ---------
------              wie diese
                                          ungeheure
Solche Überlegungen           können uns die unendliche



                                        ahnen lassen
Manigfaltigkeit der Mittel unserer Sprache zeigen      ;



                                  sich uns hier zeigt
                                 --------------
                                          ----------
                                          beobachten
           interessant mit
und es ist merkwürdig, mit dem was wir hier sehen



--------------------------------
die einfachen und starren Regeln zu vergleichen,



was
---
die Logiker vom Bau aller Sätze gesagt haben.
```

This is a plain text representation of what is happening on paper in the manuscript. For instance, a deletion is represented through a sequence of '-'-symbols just above the text to be deleted, akin to what is happening using pen and paper. Similarly, an inserted

text is placed above the position where it is to be inserted. In contrast to pen and paper however, in this electronic version, the text to be inserted creates empty space just as long as the inserted text, on the line below. The equivalent notation on paper is often a $\bigvee$-symbol that indicates the insertion's final position. This is necessitated through the material nature of paper, where we cannot simply create new physical space.

The format is also powerful enough to represent open variants. Open variants are stacked above each other and synchronised horizontally by adding more space where necessary.

The visual aspect of this format represents all this information in an obvious way but does not lose any of the complex information contained in the facsimile. The straightforward nature of the notation and its visual simplicity are human-centric: the notation is cognitively accessible to anybody, not just to computer scientists. In addition, it does not pre-structure the reconstruction of the text in such a way that it would restrain other, diverging interpretations. This schematic representation turns out to be a part of the topological notation format which will be developed in this thesis.

There might be other methods to represent the manuscript with high fidelity to the original. Another representation that is particularly suited to the non-linear nature of the text are graph structures. For instance, a *variant graph* can fork and merge to encode alternative readings of a multi-variant text, as figure 1.2 shows.

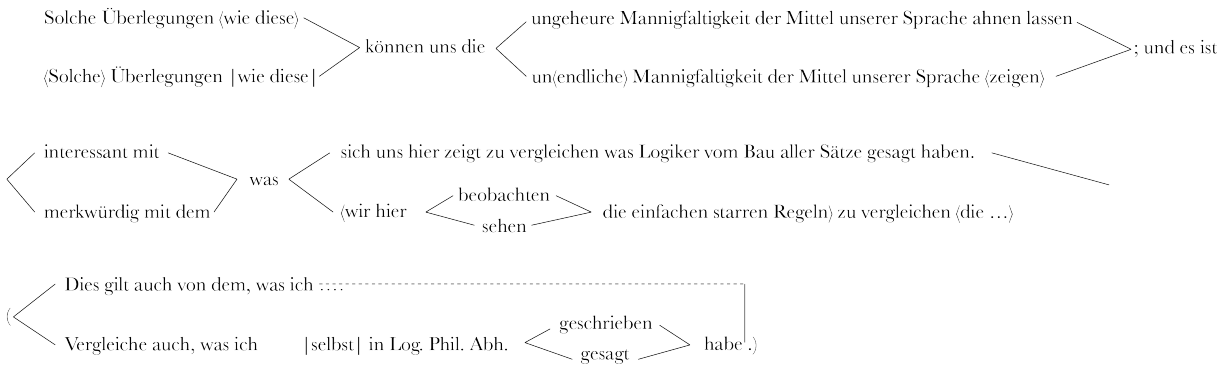


Figure 1.2: Example graph structure of multi-variant text (German)

I will now summarise what the look at Wittgenstein's manuscript has taught us about what kind of format we need.

Requirements

If we want to capture textual phenomena such as the one presented above, we need a format that is machine-readable yet intelligible to humans, interoperable between computer systems and suitable for long-term archiving. Machine-readability of a document

is generally understood as the ability to be readily processed by computers; specifically, by generic processing and manipulation methods such as validation, transformation, exchange, and other forms of document automation.

To be able to survive technological changes, shifts or even breakdowns, the desired format should further be independent of any concrete computer architecture.

Seen from the working philologist's viewpoint, the system should be cognitively accessible and readily usable for scholars in the humanities, without the need for a stand-by computer scientist. Concerning the phenomena that can be expressed in the format, it must be able to deal with overlapping structures, non-contiguous structures, multiple document hierarchies, as well as structural changes to the document. I will argue later that all these cases can be seen as the result of different types of *concurrency*. Furthermore, the system in question should also cover parallel texts such as the open variants in Wittgenstein's manuscript. Ideally, it should represent all these structures natively, in a programmatic and transparent way, without having to resort to workarounds.

Because one of the scenarios is technological breakdown, the system should be cognitively accessible without needing any machines. When designing the format, we should therefore work closely with the human cognitive apparatus, for instance by making sure that data and its storage mechanism are visually interpretable and thus as self-explanatory as is possible, without the need for a complex manual. This is one of the safest ways to insure that true longevity is built into the storage of our valuable cultural ideas.

In this thesis, I will present a document model which goes far beyond the use case of philologists. My document model is designed to be a universal document model, for all textual sources, including non-linear ones and those with multiple hierarchies. In fact, the problems that I pointed out affect many documents we deal with on a daily basis: virtually all documents in gestation are inherently non-linear. Only the last final version of any text is truly linear and can therefore be fully supported by current technology.

Outlook

It is my belief that in solving the problem discussed above, computer science and information technology can gain valuable insights for their own field from philology, with its centuries-old experience with texts. In this thesis, I will introduce structures beyond trees together with topological notation. In my system, the original ideas are clearly separated from the annotation, using spatial layout as one of the tricks that the human brain can easily interpret. My solution works with the cognitive apparatus of potential future heirs to the cultural heritage, assuming that their apparatus is similar to ours. My model

enables access to the ideas in an interoperable way across different users and different projects; with it, ideas can be woken up, played with, and frozen for safety if necessary. By providing methods for making this possible, I hope to help advance document processing.

The thesis is structured as follows: The next chapter, 'Related Work', will describe the status quo of handling complex document requirements from related disciplines. I will analyse different techniques and solutions currently in use, and will point out where and why these solutions fall short of fully meeting the requirements set above.

Chapter 3, 'Design', is the core of this thesis and presents the main parts of my idea. Here, I will introduce the ideas behind the topological notation format, describe different forms of concurrency which are necessary for the data model, and present an extended, philologically-inspired view on processes of textual revision. I will also describe how annotation works in this new format, introduce a region algebra formalism for the programmatic querying of overlapping annotation, as well as the indexing of both annotation and text. Finally, the chapter will close with a description of the data model as well as possible transformations to other data structures for various purposes, like, e.g. variant graph structures to build a full-text index.

Chapter 4, 'Implementation' explains the implementation and inner workings of the data model from a technical perspective, from building the model to crafting indexes and querying. I will also describe how a complex written artefact was ported into my document model, namely the Wiener Ausgabe (Wittgenstein and Nedo 1993–2023), a printed reconstruction of Wittgenstein's writing in his notebooks. The thesis will conclude in chapter 5 with a summary of this document model and its benefits, and point towards future developments.

Chapter 2

Related work

Chapter 1 identified two essential qualities for the successful long-term preservation of information, namely interoperability and longevity. I also presented the challenges of representing processes of textual revision in complex document requirements. Given these *sine qua non* qualities of sustainable archiving, I compiled a catalog of necessary requirements towards any recording system. Clearly, the tasks of reconstruction, preparation, data curation, recording, archiving and dissemination should be done digitally. However, if we set the printed book as a gold standard, it is not obvious what an equivalent digital solution would look like, and whether it even exists.

In the current chapter, I will study methods of information recording ranging from book printing (section 2.1), via the tree model (section 2.2), with standoff notation (section 2.4) as well as version control (section 2.5) and collaborative editing (section 2.6).

In the final section of this chapter, section 2.7, I will conclude that in the present state of affairs, no comprehensive digital solution exists that can cover all these requirements as a whole, but that insights for an integrated solution can be gleaned by compiling the advantages and disadvantages of existing methods.

2.1 Long-term archiving in print

Since its invention by Gutenberg 500 years ago, the printed book has remained of unmatched quality in terms of longevity and interoperability. This is in stark contrast to the ephemerality of today's digital files. From a long-term archiving perspective, a printed book always remains perfectly stable and accessible without additional provisions. I argue that it is a good idea to build a new document model on the basis of the expertise of philology with all aspects of manuscripts.

No matter its age, one can simply take it off the shelf and it is ready for use. From the physical perspective of the medium, the printed book can easily guarantee a comfortable shelf-life of centuries.

From a usability perspective, a book can be readily interacted with at all times, even by non-professionals, and without specialised tools or training. One can open and read it, annotate it, index it, compare it with other text pieces, and comment on its margins. Using pen and paper one is free to write between the lines or in the margins and thus one is independent from a line-based orientation or other mechanisms of the digital medium; insertions and deletions can happen freely and do not need version control; annotation is natural and independent of document structure. One can even perform cut, copy, and paste operations (at least in theory). In sum, when it comes to information preservation and usability, the printed book has a proven track record, which it acquired over centuries.

Secondly, print is stable, which is of great value for long-term archiving. By being fixed in a certain state, manuscripts also comfortably allow for an evidence-based, fine-grained and unprejudiced study. Using pen and paper, one always leaves a trace when editing, which allows careful reconstruction of the document's gestation. In contrast to this, deletions in an electronic document are done using the backspace key, which in most situations will go unrecorded and is destructive.

The preservation qualities of the printed book are likely to be rooted in way how it stores, presents, and structures information. A closer examination of these aspects could therefore get us further in our design of the ideal electronic preservation format. Beyond the purely physical structure of a printed book (the size and material of its pages, for instance), one can notice that there is a form of information structuring at the document level that is carried out through typographical conventions and spatial organisation. A headline is unequivocally signalled as a headline, a paragraph as a paragraph; and this information is immediately apparent to any reader. However, there is structure in a book that goes beyond external structure such as its separation into sections. There is primary information in a book (the main text), and there might also be secondary meta-data or further commentary. Examples for such secondary meta-data are notes by the editor or translator, or scientific citations. If secondary meta-data is present, then it is clearly separated from the running text, rather than mixed with it. Anything that is not part of the main text is strictly relegated to a separate spatial area; be it in the footnotes, be it at the end of the chapter or at the very end of the document, altogether after the main text. All that is visible in the main text is some small visual indication that some secondary meta-information is present; the reader must then seek out this additional information herself.

From a computer science perspective, the printed book uses a *standoff notation* for any type of meta-data. This arguably might be one of the key ingredients which gives print its superb usability and archivability qualities.

Let us now look at an example. The Wiener Ausgabe (Wittgenstein and Nedo 1993–2023) is a *Scholarly Edition* (together with its equivalent, called Digital Scholarly Edition or DSE), published by Michael Nedo. Traditionally, scholarly editions of manuscript artefacts have been produced in print, although some recent developments publish them only electronically (Pichler 2010) or in hybrid editions (Fanta 1994; Fanta 2007, Schnitzler 2018). The Wiener Ausgabe is a multi-volume print edition of Wittgenstein's written estates, reconstructing the genealogy of his work (cf. the stemma in figure 2.1). The ideas in this thesis are strongly influenced by Michael Nedo's experience from 40 years of publishing the Wiener Ausgabe, together with his technical vision for how electronic archiving should work.

Wittgenstein kept re-writing his remarks over several stages and in intricate ways, starting from small pocket-sized notebooks, and developing them into larger manuscript books.

The remark (German: "Bemerkung") is Wittgenstein's basic unit of writing. The latter were collected into cleanly-written typoscripts (by typists[1]), from which Wittgenstein made cuttings ('Zettel'), which he then sorted through and re-arranged into the versions that should become his planned books. Figure 2.1 shows the genealogy of these writing processes across documents, called a *stemma* in philology, whereby the top represents the origin and the bottom the final results. The individual documents were given specific names by the philologists (e.g. 153a, etc.) to tell them apart. For instance, Wittgenstein wrote initial remarks in his pocket notebook 153a. Later, he would revisit these and incorporate and re-write them in other places, such as manuscript book 110, 111, 112, and 113. A year later, Wittgenstein collected all remarks from manuscript books 109–114 and has these typed into typoscripts by typists[2]. Note how Wittgenstein is yet again unsatisfied with document 213, *The Big Typescript*; after all the concentration that went into it, it explodes again into other revisions. (And in the end it will never be finished.)

Although the Wiener Ausgabe is a print edition, for obvious reasons it is prepared electronically. The print edition of the book collection is compiled using a specialised software package, the WittgenEd software, a self-developed, custom markup language for the recording of writing processes.

[1]Sidenote: the philologists working on the Wittgenstein estate can even tell different typists apart by a close analysis of the ink cartridges used on the specific typewriters. This almost detective-like manner shows the attention paid to this philosophical estate by the philologists.

[2]Sidenote: by this time, Wittgenstein was so intimately engaged with these remarks that he didn't even consult his own writings but instead largely dictated them from memory.

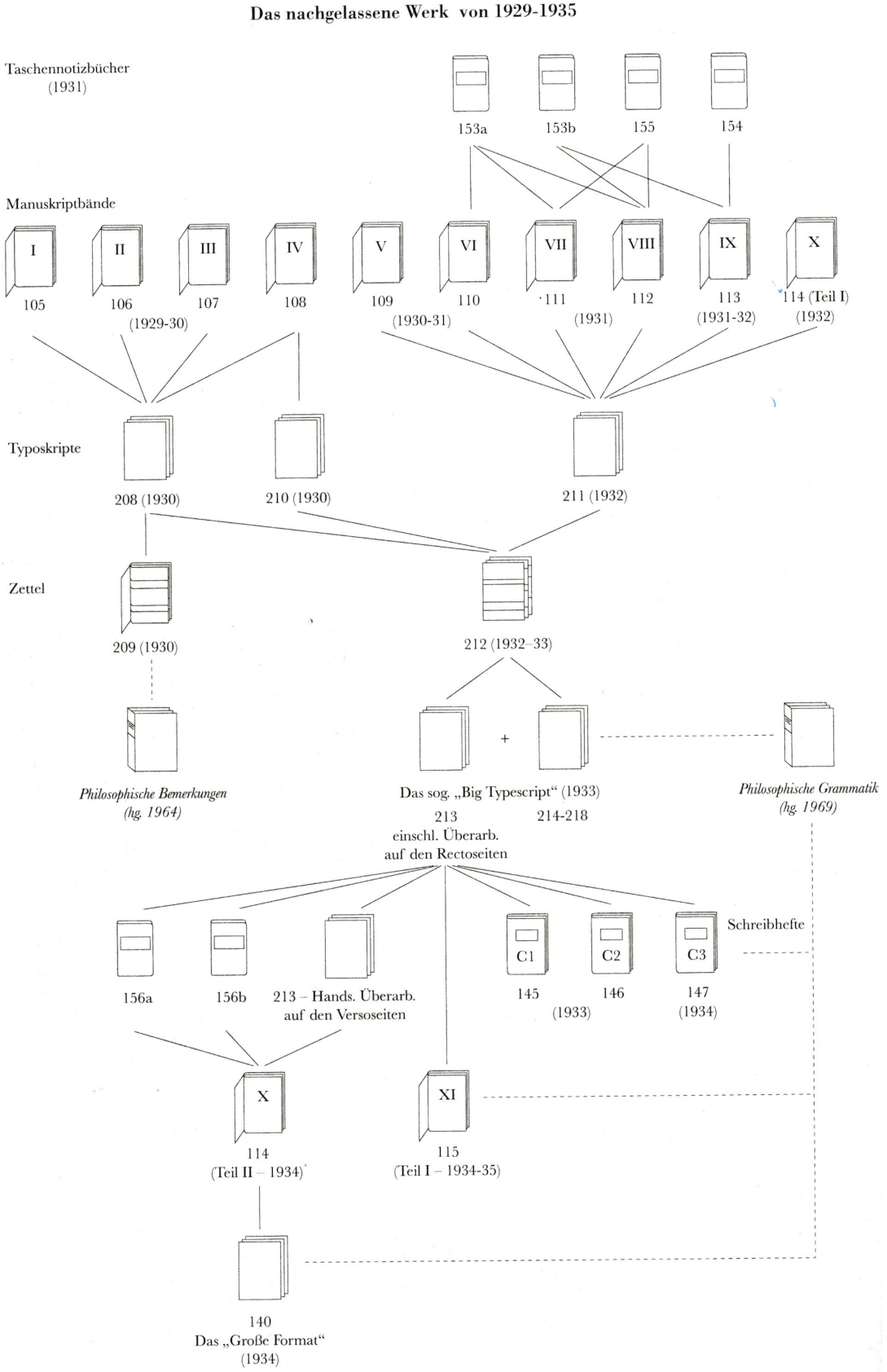


Figure 2.1: Stemma of Wittgenstein's works (taken from Wiener Ausgabe)

The internal WittgenEd markup language was developed to cater to the specific needs of the editors as well as the intended print layout. New typographical conventions were invented, including diacritic marks and other means, and these play a major role in the recording of textual revision processes. The typographical conventions need to accommodate the non-linearity of the primary text, as well as the extensive commentary explaining the primary text. This typically takes the form of extensive annotation and commentary which explain findings.

The internal markup language is intentionally kept to a small and precise set of so-called "mnemonic" codes, so as not to pose additional cognitive burden onto the editors during their complex reconstruction work. In effect, the internal WittgenEd markup language represents a classic markup language with opening and closing tags (detailed example to be shown on page 113).

If the typographical conventions are not enough to represent the interpretation the philologist has in mind, or if there are unclear passages in the original manuscript, extended footnotes can be used. Typically, this is done using small, numbered indicators to the footnotes within the primary text as commonly known. If one prefers not to interrupt the reading flow, it can also be done with numbered lines in the margins of the page, which the footnotes can refer to. Of course, such external commentary must be linked back to the text it refers to.

If the information is too large or too far removed from the main writings to be put into footnotes, such commentary is altogether relegated to a specific *apparatus* volume. Apparatus volumes are created by philologists to add further relevant information to the primary text. Examples for such correlated information are an author's diaries, letter correspondence or biographical information.

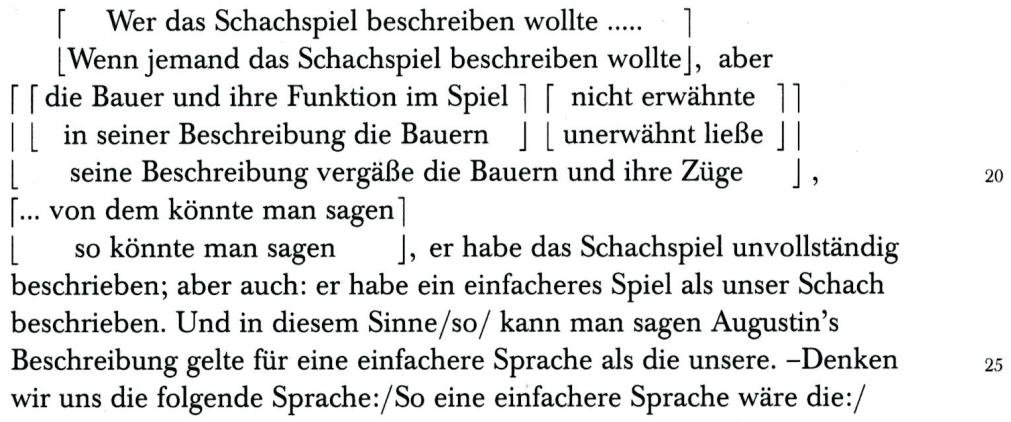


Figure 2.2: Variant text in print

Let us take a look at figure 2.2, which shows a snippet of text from the Introduction to the

Wiener Ausgabe of Wittgenstein's works (Nedo 1993). It is a prime example demonstrating how multi-variant text can be dealt with in print. Parallel, variant readings are expressed in two ways: for larger alternative sections, square brackets are used, whereas smaller expressions use diacritical marks of `/.../...`. On the right hand side of the figure, we can see numbered lines that footnotes could refer to if applicable.

The visual display of the text and its surrounding information breaks up the strict linearity of the text. This allows us to immediately read off different possible combinations of the textual variants, corresponding to the potential readings that the text allows. Note that the notation has a close resemblance to music notation, where parallel voices are synchronised with each other through spatial orientation.

This type of ergonomic information organisation and presentation results in the impressive qualities of the printed book. But these advantages come with a few major restrictions: their cost of production, immutability, and absence of machine-readability. Firstly, the replication and dissemination of such information should result in wide-spread, democratic access for educational and leisure reading purposes, but the cost of production can present a severe obstacle to this.

Secondly, once printed, errors are difficult to correct. Similarly, collective efforts of working together on a text are difficult and expensive if one does so via print. The same is true for keeping multiple, synchronised versions.

Lastly, while the printed book has excellent ergonomics for intellectual engagement, many ordinary tasks are clearly better done digitally. Examples of such tasks are indexing and search, as well as federated efforts in working together with many people in different parts of the world. Ideally, a digital library should combine the best of both worlds: the analogue world of the printed book, its longevity and interoperability, as well as the digital world of simple replication, democratic dissemination and machine-readability. In this thesis, I set out to provide tools to improve the creation of Digital Scholarly Editions.

2.2 Tree-structured document models

We have seen in the last section that typographic conventions in combination with human interpretation are powerful means to structure documents and to convey ideas. However, if one wants to make digital documents machine-readable, the intended semantics of a document must be made explicit. To do so, markup languages are used. A markup language lets the user declare a headline as a headline and a paragraph as a paragraph in an explicit manner. This is a precondition to machine-readability and interoperability.

Different markup languages exist, and their syntax can look very different. For instance,

when using LaTeX as the markup language, one would express emphasis using a notation `like \emph{this}`, whereas the same would be expressed as `like <emph>this</emph>` in XML.

While markup languages can differ in their syntax, what all markup languages have in common is that they use embedded markup. A structured document with embedded markup consists of a primary text together with structural meta-data, embedded right into the primary text itself. To separate the structural information from the primary text, specific markers are used, often referred to as *tags*. These language-specific markers circumscribe the respective text region from begin to end, thus explicitly structuring the content.

In all cases, however, the central structure-building mechanism behind the markup is the hierarchical data model of the *tree*. A tree is defined as a directed acyclic graph (DAG) with a single root and the rule that every node can only have a single parent.

A directed acyclic graph structure consists of a set of nodes (often also referred to as states) and a set of edges. In a DAG, nodes are connected through directed edges, such that following the paths of these edges will never lead to a closed loop.

The fact that XML documents represent a tree is the central provision that guarantees the machine-readability of the document. The tree structure itself is expressed via context-free grammars.

A document must follow specific rules in order to qualify as a structured document. For one, tags must come from a given vocabulary. The vocabulary of available markers can either be a fixed set provided by the specific markup language used, or, if the language allows, can also be a user-definable exchangeable set described via a separate document, the document schema. DTDs (Document Type Definition) are one form of document schema.

Secondly, regions of markup cannot arbitrarily be spread across the document, but must be correctly nested to build a *well-formed* document. Well-formedness of a document mandates that opening and closing markers of any tag cannot open and close arbitrarily, but must be paired such that markup regions may not overlap and must be fully contained within each other, obeying the hierarchical organisation of the tree data structure. In practical terms, as we follow the text linearly, this means that each region that is not closed yet must close in the reverse order of its opening. If this applies, then the document is well-formed and thus, machine-readable.

Lastly, even within a well-formed, hierarchical organisation of the tree data structure, tags can still not be used arbitrarily. They must fulfill a second condition beyond

well-formedness, and that is *validity*, adherence to the grammatical rules given by an associated document schema. If a document is valid, it can be *validated* by a process called validation. The purpose of the validation mechanism is to check the document's structure for consistency above and beyond mere well-formedness.

One of the most widely used of markup languages is XML, the eXtensible Markup Language. XML evolved from SGML, the Standard Generalised Markup Language, of which it is a reduced subset.[3] In XML, a *schema* is used to specify the document's type. A schema specifies the available tag elements, as well as their allowed use within the hierarchical organisation of the document structure. The grammatical rules defined in the schema might declare that headlines cannot appear in footnotes and subsequently these rules can be used to validate the document's structure. Listing 2.2 shows an example of sentences grouped together as a paragraph.

Listing 1 Well-formed XML

```
<paragraph>
  <sentence>The quick brown fox jumps over the lazy dog.</sentence>
  <sentence>The quick brown ferret jumps over the lazy dog.</sentence>
  <sentence>The quick brown fox jumps over the lazy pig.</sentence>
</paragraph>
```

The data model of the tree represents an ordered hierarchy of elements. Elements in this recursive data structure are referred to as parent and child nodes, corresponding to the structural elements of the document organisation. Starting from the root node, any node can have arbitrarily many children. Child nodes are ordered sequentially and each child node can again have optionally many ordered children. The final child nodes of the tree, which by definition don't have any children of their own, are referred to as *leaf nodes*.

Within the sequentially ordered organisation of the tree, one can observe two basic dimensions:

- one vertical dimension of *hierarchical dominance* between parent and child nodes (represented by a relation *contains* / *is-contained-by*)
- as well as one horizontal dimension of *order* between the sibling-nodes of a parent node on the same level (i.e. a relation *next-child* / *previous-child*).

Before I move on, I want to turn to general computer science terminology to explain some of the underlying assumptions between XML and structured documents. There exists a central connection between *model*, *syntax* and *validation* in structured documents.

[3] Per se, SGML is not a markup language, but a meta-language, in that it represents a standard for defining markup languages for structured documents.

The document model of a structured document is expressed through a certain syntax. As we have seen, XML uses a different syntax than LaTeX, as does JSON. The data is captured in the document model; the syntax expresses it as a linear text. Once the syntax is known to any other machine, it can reconstruct the document model and thus, the actual data, from it. We call the process of turning a linear text into a document model *deserialisation*, the reverse process of writing out a document model as linear text through *serialisation*.

In hierarchically-structured documents, the document model represents a tree structure. The data model implements the actual data structure below the document model. The document model acts as the programming interface for the interaction with the data, e.g. in the validation or transformation of data.

Well-formedness in the syntax is a precondition for successful deserialisation of the data into a document object model. Validation operates on a deeper, semantic level and checks the document model for compliance with the given document schema. Once a document is validated, it becomes available for interaction and transformation processes associated with the document model.

Let us now look at how the model-syntax-validation paradigm works in XML. There exist two major ways to deserialise an XML document. An XML parser reads an XML-formatted, linear text and returns a document object model (DOM), which is a document model. Unless the document is well-formed, the parser will return an error. Two general types of parsers exist:

- A DOM parser reads a document as a whole, returning a tree data structure, the DOM.
- A parser with a SAX interface receives a sequence events, each of which provides a relevant XML status change (such as the closing of a tag, or the reading of an attribute). This allows the programmer to construct their own DOM (or rather filtering out those parts of the tree that they are not interested in).

Once the DOM structure is available, validation can be applied. In the case of XML, validation checks for compliance to the context free grammar rules expressed in the schema. The validated DOM structure then allows programmatic document processing and automation mechanisms, namely *a)* access to arbitrary subtrees, and *b)* transformation of subtrees.

Access to subtrees can be accomplished by way of XPath, the XML Path Language. The XPath language represents an algebra for the programmatic access of sub-trees in the recursive data structure of the tree. Using XPath expressions one can navigate the tree structure and select individual nodes of the tree. An additional specification, XPointer, is

a formal method of specifying elements in the tree in a manner that is more fine-grained than XPath, down to character positions inside elements.

Given the controlled access to sub-elements of the tree structure, the XML paradigm also allows us to transform these subtrees into different subtrees, using XSLT (Extensible Stylesheet Language Transformations). XSLT is a functional programming language, that allows the programmer to specify transformation rules for sub-trees.

The entire package of validation, access and transformation that XML offers is a powerful way of dealing with hierarchically-structured documents. Using the XML suite of tools makes it possible to treat XML documents as a form of a database. Input and output of processes are precisely defined and there is a correctness guarantee.

A few corollaries follow from the above definition of the tree structure. For one, the tree model of XML enacts a single hierarchy (also called a mono-hierarchical structure), where each child can have one and only one parent node.[4]

This is necessitated by the fact that elements need to be nested correctly. Next, there cannot be any overlapping regions.

These two restrictions together run counter to the ambition to describe the multi-dimensional nature of a born-analogue document. For instance, if one wants to describe a manuscript page along its material boundaries of page dimensions, together with a textual description of lines and paragraphs, it already becomes theoretically impossible to describe this for the general case using only a single hierarchy of non-overlapping elements. Special cases exist where a single hierarchy is enough, but in practice, paragraphs do not always end at the bottom of a page, sentences flow over lines, and words sometimes need splitting across line ends. Thus, there is already a clash even if all we want to describe are page dimensions as well as the linguistic elements contained on these pages, such as words, sentences, or paragraphs.

The impossibility of having two annotated regions overlap each other is a well-known problem, sometimes referred to as "the notorious overlap problem". While some authors see it as the main problem with tree structures, in our DH scenario, other shortcomings of the tree model exist and are even harder to deal with. Philologists may need to define non-contiguous structures (for instance to express coordinated editorial editions), something which XML cannot do. They may also want to record changes to the document's hierarchy itself during the gestation of the document, which again is beyond XML's capabilities.

[4]Besides many other features, XML's predecessor, SGML, did have a `CONCUR`-feature to allow for more than one concurrent hierarchy; yet historically, this feature was only ever specified but never fully implemented. When XML, as the simpler subset of SGML, was defined, its creators decided to constrain it to one single hierarchy.

Such situations are usually dealt with using *workarounds*, which circumvent the formal tree definition while keeping the data well-formed enough so that it is still (just about) a tree. Workarounds are hacks that may be necessitated by the use case, but that nevertheless go against the spirit of XML's tree model. Their effect on the machine-readability of the document is often disastrous.

Consider for example validation under XML. In general, we assume that if a document complies to its schema, it is semantically interpretable and valid. Workarounds, for instance those necessitated by multi-hierarchically structured data, destroy this type of guarantee. In this case, XML validation has been short-circuited. Even if a document formally "validates", this type of validation is no longer meaningful.

The use of workarounds to deal with the fundamental limitations of the tree model are common in today's DH landscape, where the tree model is in widespread use. Despite feeling the negative effects of this situation on a day-to-day basis, practitioners rarely question the applicability of the tree model. I therefore feel compelled to sketch out in more detail the problems, their workarounds, and the disastrous effects of the workarounds on longevity; I will do so in the sections that follow.

2.2.1 Overlapping hierarchies

Arguably the most common case where the tree model falls short is in the case of overlapping annotation. In order to make overlapping structures still work within the mono-hierarchical organisation of the tree model, this section will discuss the most prominent of workarounds, the so-called *fragmentation method*.

Listing 2.1 schematically exemplifies a case where one would like to annotate a linear text with two overlapping pieces of information, namely one in which a certain part of the text is underlined in red and another in green.

Listing 2.1 Parallel overlapping annotation

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit
`---------' underline red
      `------------------' underline green
```

In order to fit these non-nesting pieces of information into the hierarchical organisation of the tree data model, an artificial fragmentation of the annotation becomes necessary (schematically displayed in listing 2.2).

Listing 2.2 Fragmented overlapping annotation

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit
`----'`---' underline red
      `------------------' underline green
```

After the fragmentation, the mandatory nesting of the tree can be realised again. However, not only does the description become convoluted, and, arguably semantically less close to the original intent, the newly separated pieces now present a different type of a hierarchy and also remain isolated, unless recombined through additional meta information.

In order to reconnect and reference the otherwise isolated tree nodes, one typically uses attributes on the tags: `comment1-1` is connected to node `comment1-2` through the `next`-attribute (cf. listing 2.3).

Listing 2.3 Overlapping annotation and XML fragmentation

```
<p>
<span id="comment1-1" next="comment1-2">Lorem</span>
  <span id="comment2">
    <span id="comment1-2">ipsum</span>
    dolor sit amet
  </span>
, consectetur adipisicing elit
</p>
```

The visualisation of the resulting structure in figure 2.3 exemplifies the intended cross-connection of nodes through a `next` attribute in the tree.

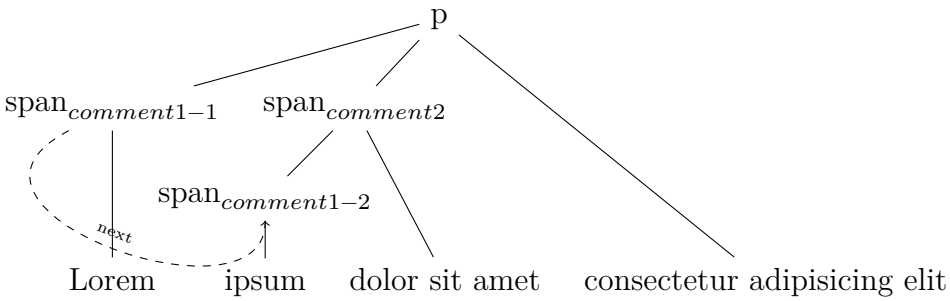


Figure 2.3: Visualisation of the fragmentation method using attributes

Figure 2.3 shows the attempt to cross-connect two nodes of the tree across different branches, thereby transgressing the hierarchical organisation of the tree model.

An alternative visualisation shows how this workaround attempts to coerce one node in the tree to be the child of two parents. The dashed line of figure 2.4 shows the attempt to make the leaf node of `ipsum` part of two hierarchies: `span_1` and `span_2`. The semantics of this structure goes beyond what the tree can model.

Notice the high dependency and tight coupling of the `next` attribute. Such circumventions of the tree model cannot be handled in the document schema and subvert the standards in document processing and automation. It is impossible to resolve the intended cross-connection of nodes in the tree in general ways and all downstream tooling is required

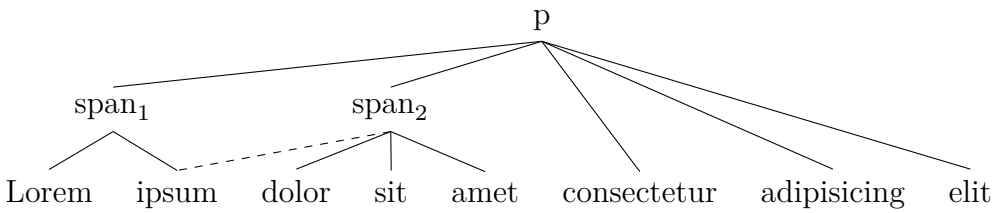


Figure 2.4: Multiple parentship for child node in the tree

to explicitly respect this special rule in order to avoid a misinterpretation of the data. Through such practice, the data becomes idiosyncratic.

Apart from splitting the source document into redundant copies with different annotations, there are basically two more workaround methods that are commonly used in such cases[5]:

- fragmentation (as seen above)
- boundary-marking through empty elements (often referred to as the 'milestone method')
- standoff-annotation from the outside by pointing at elements (and into the tree)

All of these methods carry problems and can introduce severe problems for all downstream tooling, impeding and jeopardising long-term preservation prospects, in an escalation as follows:

- In the fragmentation example, the formal tree model is violated by some nodes now having two parents.
- By transgressing the tree paradigm, the data becomes ambiguous, side-effects across the tree's nodes are introduced and the document is unavailable to proper validation and document processing.
- This ambiguity introduces a dependency on all downstream tooling for all further document processing and automation.
- Introducing such ambiguity makes the data *idiosyncratic* in that any tooling must take special precautions to deal with the data accordingly. The data has effectively changed its nature from context-free to context-sensitive. In the context of a node with two parents, the parser must remember which parent it came from.
- What is even worse is that these workarounds dupe validation mechanisms into accepting the data as well-formed. *Syntactically* the model is well-formed and can even validate but *semantically* it may be corrupt or lead to unintended side-effects.

[5]https://www.tei-c.org/release/doc/tei-p5-doc/en/html/NH.html

- In general, the additional meta-data to recombine the artificially fragmented nodes makes the data more complex and bloated; the complexity increase over time will be exponentially difficult to handle.

2.2.2 Non-contiguous hierarchies

Non-contiguous structures are pieces of information that are part of different dominance relations within the tree, but which are related in some way. If these are present in the data one wishes to model, again the tree model is unable to adequately model the situation. To consider this situation in detail it is helpful to revisit the example from the introduction.

> *[Such] considerations /like these/ can {show us/let us fathom} the {infinite/vast} multitude of the means of our language; and it is {odd/interesting} to compare with what we {see/observe/what is revealed to us here}, the simplistic and rigid rules which logicians have proclaimed about the structures of all sentences.*

We have seen that there are two cases of variant readings introduced by Wittgenstein. One pair is *{show us/let us fathom}*. The other is *{infinite/vast} multitude of the means of our language*. Since both text pieces are at different places in the linear structure, this can result in different interpretations.

One might choose to accept all possible combinations, resulting in 4 different linearisations. Alternatively, one can interpret the two variations as connected, resulting in two distinct readings of the sentence. Editors need mechanisms to do this in their work of editing a text vs. transcribing it. As discussed earlier, such an interpretation might correspond to a shift in understanding the productivity of language from a mathematically-precise concept to an intuitive one.

Listing 2.4 Non-contiguous structures

```
<remark>
  Such considerations can
  <choice>
    <variant>show us</variant>
    <variant>let us fathom</variant>
  </choice>
  the
  <choice>
    <variant>infinite</variant>
    <variant>vast</variant>
  </choice>
  multitude of the means of our language
</remark>
```

Listing 2.4 shows the non-contiguous elements concerned. In case one interprets the two editorial operations as coordinated and wants to accordingly express this in the tree model, one would need to connect them somehow.

As we have seen in the fragmentation workaround, one could link the respective nodes through the use of attributes. However, this would result in the consequences already discussed above. Again, such links would reach across the tree's boundaries and make some nodes the children of two parent nodes, violating the tree structure. Figure 2.5 illustrates the intended cross-linking of nodes across hierarchical boundaries.

We already know the heavy price of misusing the tree model. XML's promise of interoperable document automation, consisting of proper (deep) validation, interoperable exchange and interchange is only available inside the tree paradigm.

Although no overlapping structure exists in this case, the document again has become idiosyncratic. Not only are idiosyncratic documents hard to understand but they require custom maintenance. To maintain idiosyncratic documents in a large DH project, a computer scientist who acts as the workaround fixer has to stay on constant standby, and write code every time anybody wants to do something new with the documents. For the interoperable long-term archiving prospects of the documents, this is disastrous.

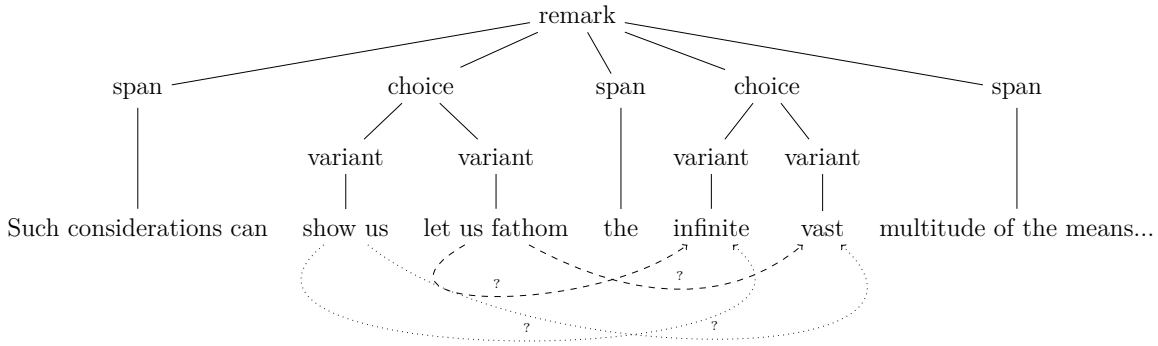


Figure 2.5: Intended cross-boundary linking of nodes for non-contiguous structures

2.2.3 Revision of the document's structure

The previous sections were concerned with workaround methods to model changes to the primary text. However, changes can also happen on the structural level of the document, for instance, when a whole paragraph is removed or a paragraph break is inserted.

In the simplest scenario, one can imagine that a user wants to split a paragraph into two. In the user interface, this could be accomplished by typing a single return character at the position where the paragraph split should occur. Evidently, this change leads to a different structural organisation of the document, as is illustrated in figure 2.6.

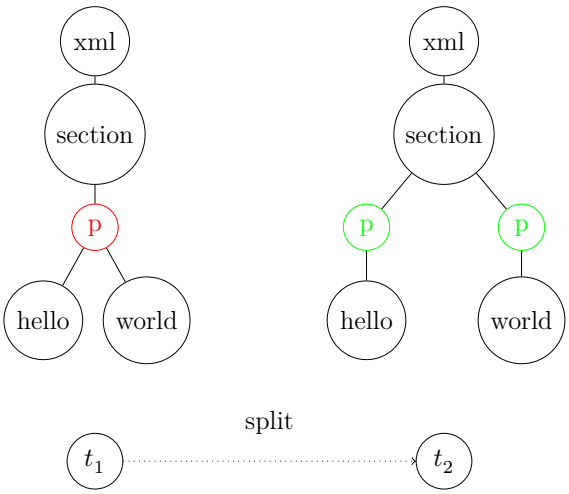


Figure 2.6: Schematic view of a paragraph split

Recording such changes is important, for instance when several people work on the same document. While a paragraph split might not be a severe change that might harmfully impact the intentions of a document, adding or removing information can be. To see that this is a case, imagine a patent lawyer and their client working together on a patent application. This is why word processors like Microsoft Word and OpenOffice offer a "Track Changes" functionality for such recording of changes to the commonly shared and edited document. This functionality is also referred to as versioning. Versioning includes the recording of each and every change by author and date, so that changes can be collated and undone at a later point in time.

How can we make the versioning of the structural change explained above work within the confinements of the mono-hierarchical organisation of the document? Again, this involves heavy use of workarounds. OpenOffice actually employs a mix of different methods: the milestone method, together with standoff-markup, which are all added to the tree as embedded markup.

Di Iorio et al. (2011) observe how OpenOffice versions these differently structured mono-hierarchical trees within the single hierarchy of the main document itself:

> *…For instance, the insertion of a return character and a few characters in a paragraph creates a structure as follows:*
>
> ```
> <text:tracked-changes><text:changed-region text:id="S1">
> <text:insertion><office:change-info>
> <dc:creator>John Smith</dc:creator>
> <dc:date>2009-10-27T18:45:00</dc:date>
> </office:change-info></text:insertion></text:changed-region>
> [... other changes ...]
> </text:tracked-changes>
> ```

```
[... content ...]
<text:p>The beginning and
<text:change-start text:change-id="S1"/></text:p>
<text:p> also<text:change-end text:change-id="S1"/>
the end.</text:p>
```

> *The empty elements* `<text:change-start/>` *and* `<text:change-end/>` *are* milestones *marking respectively the beginning and the end of the range that constituted the insertion, while the element* `<text:insertion>`*, before the beginning of the document content, is* standoff markup *for the metadata about the change (author and date information).*[6]

On the one hand, all we did was enter a single character. On the other, the resulting effect (a paragraph split) created a disproportionate increase in complexity in the tree model, and was only resolvable by abuse of the tree model's assumptions.

I have shown three situations in which the tree model cannot deal with real-world situations encountered in a DH situation: overlapping markup, non-contiguous markup and structural changes. It is possible to see all three as instances of *concurrency*, the general phenomenon of parallel structures. The idea of concurrency will be a thread going through this thesis. In chapter 3, I will expand on this idea, and show how the various kinds of concurrency can be united under one framework.

2.2.4 The Text Encoding Initiative

I have so far concentrated on the data model, which is a necessary precondition for long-term archiving. Another precondition is a harmonised tag set.

In the Digital Humanities, the Text Encoding Initiative's *TEI Guidelines* (TEI Consortium 2016b) are the central piece in the standardisation of tag sets for philological annotation. Although not a standard by definition, the TEI Guidelines nevertheless are often regarded as a 'de facto standard'.

The TEI's goal is to enable the interchange of tools and data between projects (TEI Consortium 2016a). The TEI stated that interoperability is the crucial factor to preservation of textual sources, and that system incompatibility is the main impeding factor to long-term preservation. Despite this finding, the TEI defines its remit only as the creation of a commonly shared, harmonised tagset. It is not directly concerned with finding suitable data models. The TEI Guidelines are presented as an abstract definition of a common vocabulary which should later be expressed in a suitable data model.

[6] Di Iorio, Peroni, and Vitali 2011

The TEI is formulated in XML[7] as "currently the most widely-used markup language for digital resources of all kinds" (TEI Consortium 2016a). The current 'P5'-version of the TEI Guidelines consists of ca. 750 elements, mainly for the philological annotation of textual sources, amounting to about 1,500 pages of documentation.

The TEI Guidelines are too large for any number of people to keep them in their head and actively work with them. The fact that they are not published in printed form admits as much; they aren't even meant to be read in their entirety. This drawback, in combination with the lack of a working data model, means that the reality is that of specialised, incompatible tooling, in different versions in many different projects. As a result, TEI's efforts notwithstanding, data and tools still are not interchangeable between people and projects. Long-term electronic archiving can still not be guaranteed. In fact, today's long-term archiving of complex textual artefacts faces the same problems as it did 30 years ago, when the TEI Guidelines were created.

As a final demonstration in this section, consider the currently used alternative to the notation I propose: listing 2.5 shows an excerpt of the corresponding XML transcription of the same facsimile presented in the introduction (listing 1.1 on page 17), taken from an existing DH project (Pichler 2010). The full transcription can be found in appendix A. Note that this already reduced transcription of the original source still requires ca. 110 lines, and is far from accessible to a human, visually or otherwise. We can also spot some of the awkward workarounds described in this chapter, e.g. a variety of non-contiguous as well as nested `<choice>` tags.

In the field of Digital Humanities, and particularly in the field of scholarly editions there is much research on integration of linguistic resources with semantic web data (Chiarcos, Hellmann, and Nordhoff 2011; Chiarcos 2012; Chiarcos, Hellmann, and Nordhoff 2012). There is also work on an RDF-based open annotation model and annotation collaboration (Haslhofer, Simon, et al. 2011; Haslhofer and Isaac 2011) and data models to support sharing and interoperability of scholarly annotations (Hunter et al. 2010). After all, the TEI is well-aware that trees are a "grossly simplified view of what text is" and, although "[it] turns out to be very effective for a large number of purposes" they admit that "[it] is not, however, adequate for the full complexity of real textual structures, for which more complex mechanisms need to be employed" (Consortium 2016a). This is why the TEI presents an abstract model, which happens to be expressed in terms of XML, despite their own doubts. It could however, at any point be expressed using a more fitting data model.

[7] originally in SGML.

Listing 2.5 XML Transcription of manuscript snippet, taken from (Hadersbeck et al. 2014)

```
<ab ana="field:PhilosophyOfLanguage_pub:W-EPB_date:19360801*-19361130*"
  emph="blbef_1" n="Ms-115,129[2]" part="N" xml:id="Ms-115_129.2" xml:lang="de">
  <s part="N" type="es">
    <choice type="em">
      <orig type="em1">
        <del status="unremarkable" type="d_c">Solche</del> Überlegungen
        <del status="unremarkable" type="d">
          <add rend="im" status="unremarkable">wie diese</add>
        </del>
      </orig>
      <orig type="em2">
        <choice type="dsf">
          <orig type="alt1">
            <c part="N" type="c">S</c>olche Überlegungen
          </orig>
          <orig type="alt2">Überlegungen wie diese</orig>
        </choice>
      </orig>
    </choice> können uns die
    <choice type="em">
      <orig type="em1">un
        <lb rend="shyphen"/>
        <del status="unremarkable" type="d">endliche</del>
        <add rend="i" status="unremarkable">geheure</add>
      </orig>
      <orig type="em2">
        <choice type="dsl">
          <orig type="alt1">unendliche</orig>
          <orig type="alt2">ungeheure</orig>
        </choice>
      </orig>
[....]
```

2.3 Graph-based document models

We have seen in the previous chapter that trees are not powerful enough to deal with the task I set myself in this thesis. The tree is a special form of a directed acyclic graph (DAG) structure, in which there must be only one (single) root node and each node can only have exactly one parent node. As we have seen, this particular restriction was responsible for the fundamental inability of the tree structure to represent *a)* non-linear text, *b)* overlapping markup, or *c)* non-contiguous structures.

From its mathematical properties, it follows that the DAG is a potentially promising data structure for my goal. Since the number of incoming or outgoing edges of a node in a DAG is unconstrained, graph structures could overcome the limitations of the mono-hierarchical tree model and its confinement to single-parent nodes. Because DAGs are less restricted than trees, they are a far more suitable data structure for our task.

For the use case of creating DSEs, the theoretical advantages of DAGs were also seen by other researchers. The earliest suggestions of a DAG-based method with a syntax was presented by Huitfeld and the founder of TEI, Sperberg-McQueen (Huitfeldt 1994; Huitfeldt 2011; C Michael Sperberg-McQueen and Huitfeldt 2000; Huitfeldt and C. Sperberg-McQueen 2001; C. Sperberg-McQueen and Huitfeldt 2008; Marcoux, M. Sperberg-McQueen, and Huitfeldt 2013)[8].

They presented the GODDAG data structure with the associated syntax TexMecs. A viable syntax was one of the requirements needed to make the data long-term archivable. Listing 2.6 shows an example of TexMecs.

To serialise their data structure and thus make overlapping structures and non-contiguous structures possible, Huitfeld et al. use embedded markup. They enhance the notation system known from XML by adding 'half-opening' and 'half-closing tags', as well as methods to assign these to different layers of a document. This results in bloated annotation which is not human readable. Huitfeld et al.'s declared goal is to produce documents that are approachable by scholars in the humanities, as well as suitable for long-term archiving of textual cultural heritage. The notations however, which are even more convoluted than the XML approaches using workarounds, are overly complex for this task. Given this, it isn't surprising that GODDAG never caught on in the DH world.

Other graph-based approaches to DSE without a syntax exist (e.g. Kuczera 2016). For his edition, Kuczera directly employs Neo4J, a graph database. For philologists using

[8]Remarkably, it was the Wittgenstein corpus that led Huitfeld to apply graph structures to complex manuscripts. See Pichler 1995; Pichler 2012; Pichler, Smith, et al. 2012; Pichler and Zöllner-Weber 2013; Erbacher 2015; R. J. Falch, Krüger, and Smith 2012; R. Falch, Erbacher, and Pichler 2013; Gradmann 2013; Hadersbeck et al. 2014; Hrachovec 2000 for the long history in this line of work.

Listing 2.6 Example of Sperberg-McQueen and Huitfelds's extended XML syntax

```
<doc|
  <p|...|p>
  <p|...
    <text|
      <front| ... |front>
      <body|...
        ...
      |-body>|-text>
      <p|Just then, we were interrupted.
      ...
      |p>
      <+text|<+body|
... |body>
    |text>
  ...|p>
|doc>
```

this form of DSE, his solution manages to provide the basic functionality (e.g. to make a digital edition of the text, to record the text's gestation through editorial operations, and to add additional meta-data to the text, for instance when a particular letter was written and by whom). But this functionality is provided by directly interfacing with the graph database. Without an independent syntax and a dedicated data model, no access to this data is available other than through the current Neo4J implementation. In the end, the data is left in the database in this vendor-specific format and can only be accessed through visualisations. If in the future Neo4J will be discontinued, all the data in this format will be lost.

Provision of data access which is independent of the medium is one of the key requirements in my list. The Neo4J approach therefore does not provide a solution to long-term archiving, at least not until the data is rescued from Neo4J in some form and transformed into a custom new data structure and a suitable syntax.

Another approach is presented by Dekker et al. Research by Dekker et al. on TAG (text-as-a graph) got under way around the time when I started my PhD work (Haentjens Dekker and Birnbaum 2017; Dekker, Bleeker, et al. 2018; Bleeker, Buitendijk, and Dekker 2020; Dekker, Buitendijk, and Bleeker 2020; Bleeker, Dekker, and Buitendijk 2021). They also came to the conclusion that graphs might be the right data structure for multiple hierarchies. They created the tagML markup language (see figure 2.7) as a syntax for their data model. Similar to Huitfeld et al., they also use enhanced XML syntax of half-opening and half-opening tags with layers, with similar human readability and long-term archiving problems. This clearly runs foul of the requirements for the current work.

As we have seen, syntax is a severe problem when we use graphs to represent complex texts.

```
[text|+D,+T,+P title="Love and Freindship" type="novel" author="Jane Austen" date=1790>
[page|D n=6>[l|D>
[head|D,T>Letter 4th<head]<l][l|D>Laura to Marianne<l][s|T>[l|D>Our neighbourhood was small, for
it consisted <l][l|D>only of your Mother. <s][s|T>She may probably have already <l][l|D>told you
that being left by her Parents in indegent <l][l|D>Circumstances she had retired into Wales on
economi-<l]
<page]
[page|D n=7>[l|D>cal motives. <s][s|T>There it was our freindship first <l][l|D>commenced —
Isabel was then one and twenty — <s]<l][s|T>[l|D>Tho' pleasing in both her Person and Manners <l]
[l|D>(between ourselves) she never possessed the hun-<l][l|D>dreth part of my Beauty or
Accomplishments.<s]<l][s|T>[l|D>Isabel had seen the World. <s][s|T>She had passed 2 <l][l|D>
Years at one of the first Boarding-schools in <l][l|D>London; had spent a fortnight in Bath &
had <l][l|D><|[del|D>slept<del]|[add|D>supped<add]|> one night in Southampton.<l]<s]
[l|D>[s|T>[q|P>"Beware my Laura<-q] (she would often say) <l][l|D>[+q|P>Beware of the insipid
Vanities and idle Dissipations <l][l|D>of the Metropolis of England; Beware of the <l][l|D>
unmeaning Luxuries of Bath & of the Stink-<l][l|D>ing fish of Southampton."<q]<s]<l]
[l|D>[q|P>[s|T>"Alas! <-q] (exclaimed I) [+q|P>how am I to avoid <l][l|D>those evils I shall
never be exposed to? <s][s|T>What <l][l|D>probability is there of my ever tasting the
Dissipations <l][l|D>of London, the Luxuries of Bath, or the stinking <l][l|D>Fish of
Southampton?<s][s|T>I who am doomed to <l][l|D>waste my Days of Youth & Beauty in an <l]<page]
[page|D n=8>[l|D>humble Cottage in the Vale of Uske."<q]<s]<l]
[s|T>[l|D>Ah! little did I then think I was ordained <l][l|D>so soon to quit that humble Cottage
for the <l][l|D>Deceitfull Pleasures of the World.<l]<s]
[l|D>[s|T>adeiu<l][l|D>Laura —<l]<s]<page]<text]
```

Figure 2.7: TAGML Example

Another problem is how to represent markup at the same time as multiple hierarchies.

One solution, albeit not fully mathematically developed one, is the hypergraph. A hypergraph is a graph that has hyperedges. A hyperedge is an edge which connects multiple nodes at once, rather than just two as in a normal graph. This is depicted in figure 2.8.

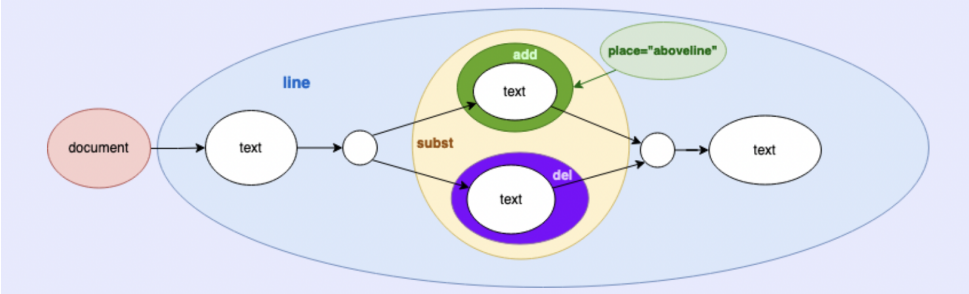


Figure 2.8: Visualisation of a hypergraph, from Dekker (2021).

Hypergraphs, if and when they are fully mathematically underpinned, promise to deal with arbirary markup on the graph, but until then this is not a realistic avenue. (Dekker et al. themselves are also in doubt about this question (Bleeker, Dekker, and Buitendijk 2021).)

For instance, listing 2.6 shows the structure of Sperberg-McQueen's extended XML syntax (C. Sperberg-McQueen and Huitfeldt 2008), while figure 2.7 shows an example of a TAGML transcription, taken from Dekker (Bleeker, Dekker, and Buitendijk 2021). Please note that this TAGML transcription itself is simplified.

There are challenges when Dekker's embedded format is parsed with a Context-Free Grammar; to solve these, so-called 'semantic actions' are used. From a language-theoretic perspective, the need to do this is not surprising, as the structures we are dealing with are context-sensitive, not context-free.

This look at graph structures shows once again the need for a better format beyond embedded markup and convoluted tree and graph structures if we want to represent complex texts.

However, even though it seems difficult to design a single graph structure expressing the entire complexity of the text so that all my requirements are fulfilled, it is possible that part of the solution for my problem can involve graphs of some form.

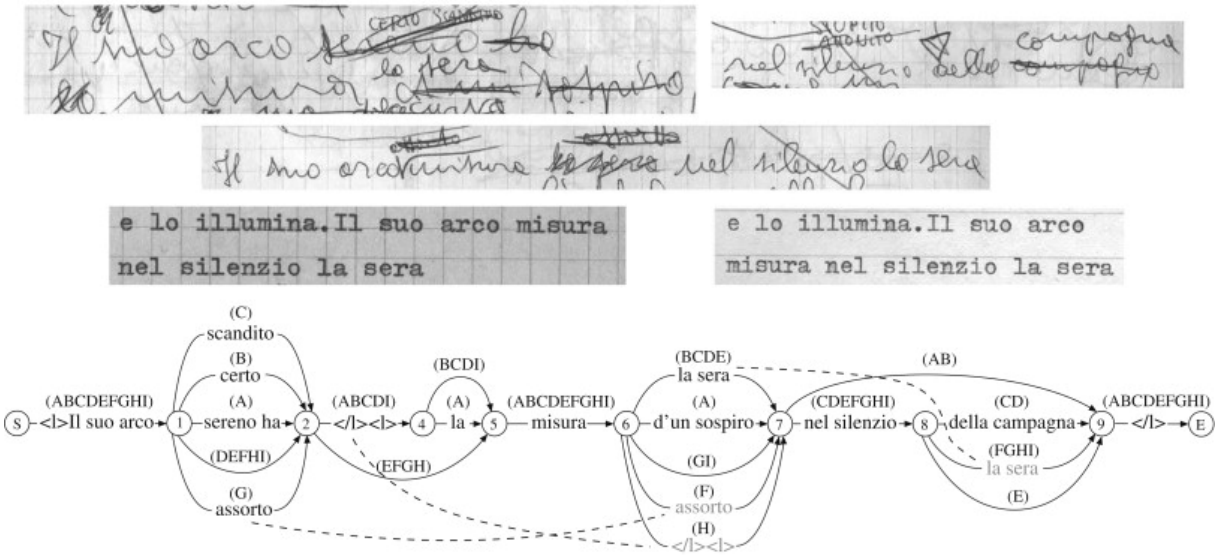


Figure 2.9: A variant graph (taken from Schmidt, 2009)

Schmidt (2009) presents *variant graphs* such as the one shown in Figure 2.9, which is concerned with representing the poem "Campagna Romana" (The Roman Countryside) by the modern Italian poet Valerio Magrelli, 1981. It models the gestation of a text from hand-written drafts to machine-typed versions.

Variant graphs have the ability to express different logically possible linearisations of a text. In variant graphs, edges are labelled with text, and nodes represent positions in the text where different variants can start and end. For instance, there are four variant text pieces between text points 1 and 2, namely "scandito", "certo", "sereno ha", no text, and "assorto". Edges are labelled with reading labels (A-I). Following the graph along edges with one reading label results in one of the allowable readings, according to the philologist who created this interpretation.

What the variant graph is able to do is to merge diverging, overlapping trees into one

single DAG, which therefore expresses a multiple hierarchy. The method by which the single-inheritance trees are merged into a graph has been called "tree-colouring", whereby a label corresponds here to a tree. After applying the algorithm, the DAG structure then represents multiple hierarchies. The algorithm works by color-marking the nodes and the edges of the individual trees; these can be combined to one single DAG structure but also be pulled apart again later (Abouelaoualim et al. 2009).

Note how Schmidt deals with annotation: in-line XML-like element markings such as `<I>` are flattened into the text. However, all opening and closing brackets occur between those text positions which are connected by a single edge, rather than multiple edges.

In chapter 3, I will employ my own version of a variant graph as an auxiliary data structure that forms part of my data model.

I started this section by saying that the idea of employing graph structures almost suggests itself for the problem at hand from a formal perspective. After studying their current use in DH, I have come to the conclusion that they introduce new problems with representing markup and with finding a simple syntax, which together make them unsuitable as the main data model I am looking for.

In addition to these structural problems, graph structures are also far more complicated to deal with than trees in practice. They require complex software libraries and complex algorithms; graph structures are always far harder to define, formalise, implement, validate, visualise and transform (Zaytsev 2015). From a perspective of simplicity they might therefore not be the best path towards long-term archiving and interoperability (at least not on their own).

2.4 Standoff document models

A completely different paradigm to the embedding of markup is the so-called *standoff*-approach. Like the printed book, standoff markup strictly separates additional meta-data from the running text and keeps both separate. As in print, meta-data is not mixed with the text but relegated to a separate section outside of the running text. The additional meta-data information is connected back into the text using pointers.

Standoff markup is a good choice from a variety of perspectives. Since standoff markup is kept separate from the main text, readers are unaffected in their reading flow. If more information is desired, one can simply consult the respective footnote. As section 2.1 on the advantages of print has shown, this approach as used in the printed book has proven to be useful over centuries. Secondly, the text does not contain inline markup and more annotation can be added at any time; we don't need to worry about making sure

that markup nests correctly and results in the right tree structures. By being freed from tree structures and workaround methods, collaborative text curation and annotation are simplified and more effective (T. Schmidt et al. 2006), and data repositories do not suffer from overtagging (Hanrahan 2015).

Standoff-approaches have been discussed in the literature in an on and off fashion (Nelson 1965; D. Schmidt and Colomb 2009; D. Schmidt 2010; D. Schmidt 2012; D. Schmidt 2014; Sanderson, Ciccarese, and Van de Sompel 2013; Barabucci, Di Iorio, et al. 2013; Waldron and Copestake 2006). There also seems to be a general agreement on the advantages of standoff markup. However, it has never gained much popularity and there is no established solution to build on. Even in the XML world of embedded markup, standoff markup is suggested only in complex document modelling scenarios (Consortium 2016b) using XPointers (DeRose, Maler, and Daniel 2000; Vitali, Folli, and Tasso 2002; Wilde and Lowe 2002; Maler and DeRose 2005). Current research on standoff markup seems to go in the direction of Open Annotation using Linked Open Data (Bizer 2009; Chiarcos 2012; Chiarcos, Hellmann, and Nordhoff 2012; Chiarcos, McCrae, et al. 2013; Haslhofer, Simon, et al. 2011; Hunter et al. 2010; Passant and Laublet 2008; Peroni and Vitali 2009; Di Iorio, Peroni, and Vitali 2010).

One of the reasons why this approach has never really gained traction in digital editing might lie in the fact that the separation of meta data and text makes a fixed and stable connection between the text and the annotation necessary. Should this connection be lost, the data is turned useless. Of course there should be means and methods to repair such situations. After all, with a bit of trial and error one might be able to interpret the data with some detective work, should the worst come to the worst. But standoff annotation on its own could never constitute the entire mechanism that ensures electronic long-term archiving, particularly not in the face of culture breakdown.

For my own approach, I find the core ideas of standoff annotation essential, because of the principle of separation of primary and secondary data. My approach will use a form of standoff, and makes sure it is supported by the right kinds of tools. I conclude that the maintenance of meta-data links into the text should be done in a dynamic form.

2.5 Diff-based Version Control and the linear data model

The obvious first choice when it comes to archiving and maintaining a history of successive versions of a document in the software engineering world is the source control management software `git`.

Setting a repository under version control with `git` allows one to go back and restore any file in the entire repository to any committed state along a timeline of revisions. In its operation however, `git` does not track the entire versions of the files in the repository, but instead only tracks an ordered succession of changes *between* different versions, organised along a directed acyclic graph of edits. These changes are represented by an operation called *diff* (sometimes also referred to as a *patch* or *delta*), which is essentially a record of insertions and deletions, together with the necessary positional information where these changes are to be applied.

In doing so, `git` is entirely agnostic towards the underlying structures of the files: `git` tracks plain-text `diff`s between any two versions A and B, such that applying the edit commands of the `diff` allows one to build file B from A. This agnosticisity is openly owned by `git`'s creators, who officially nick-named it the "stupid content tracker" in the man pages. In essence, `git`'s document model is that of unstructured, linear text.

I will now describe here how `git` achieves versioning. The computing of differences employs a given `diff`-algorithm to identify a minimal set of necessary editorial operations (i.e. insertions and deletions) which allow one to get from state A to state B. Different `diff`-algorithms exist which can produce mathematically correct sets of edits to build file B from A. There are sometimes parallel sets of edits, corresponding to different ways to get from A to B. Some of these sets of changes comply more with human intuition than others. While all published `diff` algorithms produce mathematically correct `diff`s, they differ in exactly which sets of changes they propose, resulting in various degrees of human-acceptibility, as recent experiments have shown (Barabucci, Ciancarini, et al. 2013; Barabucci, Ciancarini, et al. 2016a; Barabucci, Di Iorio, et al. 2013; Barabucci 2013; Barabucci, Ciancarini, et al. 2016b).

But if `git`'s linear document model is applied to hierarchically structured documents, the application of `diff` can give results which are dissatisfactory. I will show this with an example. Listing 2.7 shows a linear `diff` of a XML-structured document, in which two paragraphs are joined. In the linear view of the `diff` format, the lines which are to be deleted are marked with a `-`-symbol at the beginning of the line (lines 3 and 4), while insertions would be marked with `+`-signs. Note that the listing corresponds to a simple paragraph join, if the document is seen as structured, as opposed to the misleading nature of this linear interpretation.

Since `git` is agnostic towards structure, and views any file as a plain-text file (treating characters that might indicate structural elements just as it would normal textual characters), the resulting structure `<p>Lorem ipsum dolor sit amet</p>` simply relays the information that by deleting a closing `p` tag and an opening `p` tag we arrive at identical

Listing 2.7 Line based `diff` on structured documents

```
  <p>
    Lorem ipsum
- </p>
- <p>
    dolor sit amet
  </p>
```

resulting files. On the one hand, this is of course mathematically correct. On the other hand, it might be perceived by a human as arbitrary in the sense that it is not capturing the true dimension of the difference, which is structural. (The two `p` labels concerned didn't even belong to the same `p` elements). Therefore, `git`'s result might even be interpreted as misleading.

The `diff` itself doesn't capture enough semantics to clearly convey the structural change, namely that of two paragraphs having been joined. This is because the `diff` only represents a syntactic difference between the two versions and thus cannot truly track the semantics of structural differences. Since Digital Scholarly Editions have the aim of precisely modelling textual changes in historic documents, such mechanisms are not suitable.

`git` technology can also be used for collaborative editing, for instance in a software development situation. One file that is part of the repository can then be edited by two different people. In this situation, `git` joins two `diff`s to form one single, coherent `diff`, which can then accordingly be versioned in the history of changes. If both parties have edited the files in distinct places, this is no problem. Has the file been edited by both parties in the same place, however, a *merge conflict* occurs. Such situations are often dreaded once a certain complexity is reached in the changes, since they can only be resolved manually. A human then has to manually craft a version including both versions, or simply choose which version should win. Most often, such merge conflicts occur between two given versions of a file, but sometimes, they can also involve more than two versions (when, for instance, more than two people are involved), necessitating a so-called *octopus merge*.

In a software engineering situation, octopus merges are rare, but in the reconstruction of manuscripts such situations occur quite often. In philology, the situation is referred to as a *collation*. Collations are needed when an author writes similar versions of the same piece in different documents (so-called *witnesses*). In that case, the philologist tries to collate these different versions to reconstruct their development. Often, such collations can stem from half a dozen to a dozen documents, ultimately resulting in an 'octopus merge'-situation.

Finally, the `diff` model of insertions and deletions as editorial operations is not sufficient for my requirements. As was shown in the introductory example, authors writing with pen

and paper often use the space between the lines to add different wordings, to be decided on later (or to be left as is, as Wittgenstein often did in the open variants). The imperative of the `diff` model is to force a decision down to one linear version. Such a strict limitation would be insufficient for what we want to achieve, as we need to model open variants for the recording of the gestation of a handwritten piece.

In sum, `git` is a useful methodology of keeping track of document changes. Ideas from it might be incorporable into my final data model, but it has two problems that would need to be dealt with: it is structure-agnostic, and it wants to force a single linearisation of any text.

Making diffs meaningful again: Simple, parallel work and versioning using git

Given the line-based format of the document model I will develop in this thesis, collaborative work and git-versioning become simple, intuitive and straightforward. As was shown above, git by its very design sees every file in linear format. For structured documents however, such diffs can be misleading. In my solution, sense is restored through the topological arrangement. The changes can then easily be glanced over and spotted, unencumbered by irrelevant information. This is a crucial part of my document model's design.

Imagine a situation where a user adds another annotation (in this case `underline green`, when there is already a red underline present). Innocent as it looks, this is a multi-hierarchy tree in the tree model and would require workarounds (listing 2.8). Here it is solved with the arbitrary-fragmentation workaround, similar to the example in section 2.2.1. The insertion of the new annotation is shown by two "+" characters, but these seem arbitrary and only accidentally related.

Listing 2.8 Diff example on fragmented XML

```
 <p>
 This
     <ul id="ul-1-1" col="red" next="ul-1-2">is a</ul>
+     <ul id=ul-2-1 col="green">
         <ul col="red" id="ul-1-2">
             simple
         </ul>
+    </ul>
 sentence with a metric structure
</p>
```

Since my model instead works with synchronised layers on separate lines, the same change could be displayed and captured beautifully with a simple line-based diff (listing 2.9). In my solution, we can then easily spot the newly added layer in the patch (the "+" on the bottom line succinctly and precisely expresses the structural change). Humans are likely to find such a diff explanatory, rather than arbitrary.

Listing 2.9 Diff Example in my model

```
 This is a simple sentence with a metric structure
      ----------- underline red
+          --------------- underline green
```

2.6 Collaborative Editing

Collaborative editing is an active area of research, involved with algorithms to enable collaborative creation, editing and refinement of documents by multiple people in real-time. Examples for this include products such as Google Docs or Overleaf. Both Digital Humanities and collaborative editing are in the business of tracking a series of changes to a base text. The challenge is to synchronise and consolidate concurrent edits by multiple users on the same (structured) document. Consolidation is not a simple task however:

a) There must be ways to resolve conflicting edits on textual level: what if one person deletes a word, while another replaces it by something else in the same moment? Is it a merge conflict? If yes, which text should be chosen as the final version? Can we resolve the conflict to a single, linear version of the text (and how?) or do we need to keep a potential powerset of different virtual documents?

b) Additionally, conflicts involving markup, rather than raw text, also exist. Proper consolidation means keeping the document properly structured at the markup level. What if one person marks a word as bold, while the other person marks the same word as italic, and both actions happen at the same moment? What if two such edits happen to overlap each other?

There are two main schools of collaborative editing algorithms: *Operational Transformations* (OT for short, c.f. Sun and Ellis 1998; Ferrié, Vidot, and Cart 2004), which is the historically older approach, and *Conflict Free Replicated Data Types* (CRDT, c.f. Shapiro et al. 2011b; Shapiro et al. 2011a; Kleppmann and Beresford 2017), the relatively younger approach. In OT, a single main document is kept, and all editing processes from all clients are registered and coordinated centrally. In contrast, CRDT does not depend on a main document to register individual edits. Instead, CRDT provides a data structure that allows multiple clients to work on their individual copies of the document, where all changes to these individual documents are collected independently. Once these documents are to be consolidated and merged back together, the CRDT algorithm establishes a common document of all individual changes, by applying changes in the correct order and at the correct positions. Thus, under CRDT, there is no single source of truth in the direct sense.

There is much shared ground between collaborative editing and philology. Both fields are occupied with maintaining, curating, collating and modelling changes made to texts. Yet the main difference between these fields lies in the dimension of time. When preparing

a digital rendition of a potentially heavily revised manuscript, philologists reconstruct a text and its gestation *after the fact*, mostly by one person, often hundreds of years after the original texts have been written. This curation work might span months or years of annotating. After storing this work, it might be years or decades until another scholar opens the record and starts working with it again. While of course synchronous collaboration between different DH projects might happen, the main worry for philologists is to keep the data available and accessible for such future collaboration. In contrast, collaborative editing algorithms deal with multiple authors making changes to a text, in *real-time*, within milliseconds of each other. In sum, collaborative editing algorithms manage synchronous changes to a commonly shared text, whereas the philologist's work is characterised by an extreme form of asynchronicity.

These different approaches correspond to two radically different conceptions of time and concurrency. In my use case, there is a strong emphasis on data curation in comparison to collaboration. Collaborative editing, with its aim of constant consolidation to a single, linear, final version of the text, is perfectly suited for collaboration but cannot support the curation aspect well enough. There are several reasons for this:

a) Philologists don't need or want a single, consolidated version of the text. Instead, they want to be able to record many parallel versions of given text pieces. Parallelity comes from various sources: possible readings of the same text, as perceived by one philologist; different expert opinions on the same text; as well as different versions of a text at different stages in the gestation process. Such textual ambiguities, differing expert opinions and textual versions in the timeline of gestations across different documents are all valid objects of philological research and thus need to be recorded. Philologists want a *parallel* text source rather than a linear one. In technical terms, they need a virtual document, corresponding to a powerset of all potential documents that could have been written by the original author.

b) Philological work, by its nature, consists of heavy annotation: the outcome of a philologist's intellectual work is exactly the annotation. It is common that the volume of annotation exceeds the volume of the original text it comments on by a large margin. A usable document model for this situation must therefore give annotation first class citizen status. Annotation in this form entails concurrent markup; ultimately, this involves overlapping, non-contiguous, and multi-hierarchical situations. We therefore need dedicated methods for concurrent text with concurrent markup and annotation, as provided by my document model.

Although there is a lot of shared ground between collaborative editing and philology, the crucial reason why collaborative editing, as it is now, cannot help with this use case

is that it is only concerned with linear sources, and it cannot treat concurrent markup; particularly in long-term data curation scenarios.

In any case, what we have seen here is more evidence that a dedicated document model is needed, such as the one I will present in this thesis.

2.7 A necessary departure from the tree paradigm

In the current chapter, I have examined the status quo of long-term archiving of textual cultural heritage as well as potential solutions from related fields. Table 2.1 summarises the key aspects of the two main document models considered thus far, the plain-text models (exemplified by git and the diff-model) and the tree model (exemplified by XML).

	Linear	Hierarchical
Document model	one-dimensional vector	ordered tree
Syntax	Plain text files	Embedded markup
Dimensionality	1-dimensional sequence	2-dimensional hierarchy: dominance, order
Data structure	linear memory with index-based access	single-rooted mono-hierarchical tree
Language / grammar type	Regular	Context-free
Access and extraction	Sequential access, regular expressions (e.g. grep)	Structured access using XPath, XPointer
Transformation	Regular expression rewrites (`s///`)	Structured transformation, e.g. via XSLT

Table 2.1: Comparison between the linear and hierarchical document model

As table 2.1 shows, both paradigms to recording text build on a document model. Each has a corresponding storage structure, along with methods for data access and mutation. Each supports modelling of a certain formal language type.

The linear document model is a 1-dimensional sequence, with all that implies. Such a document model has only linear memory; whichever data structure we use, the only access provided natively is sequential access. If we want fast access at search time, this will

require the previous compilation of an index structure. When accessing and extracting information from this type of data model, we can only work with regular expressions, not with context-free expressions like we were able to in the tree model. `grep` exemplifies this type of access. If we want to transform data, we can again only do so with a regular process with rewriting, such as the command `s///` in scripting languages, such as `awk`. In sum, the linear document model employed by `git` is starkly different, in remit and operation, to the tree-based document model. In the linear model, there is no structure to the document, other than a character's position. While one can conventionally agree to interpret text in the middle of a line as a headline, such extra-textual agreements are not sufficient for interoperability and interchange of data between projects. To make such structural information machine-readable, a formal grammar would be needed.

The ordered hierarchical tree model with its recursive nature, in contrast, has the power to represent such structure. Structured documents are defined by a context-free grammar, which usually is expressed through embedded markup. In the case of XML, extraction of subtrees can be performed using XPath, whereas transformation of subtrees can be done using XSLT (Extensible Stylesheet Language Transformations).

However, as we have seen in this chapter, neither the tree model nor the linear text model is able to provide adequate solutions for our situation, despite the respective capabilities of the two document models. Both document models fall short in the context of concurrent annotation over concurrent text. Additionally, collaborative edits can happen, addding a further dimension of difficulty.

The expressiveness of the linear, plain-text document model is far below than of the tree model. We have seen in section 2.5 that it is an inadequate document model for my purposes: access is only linear; there are no correctness guarantees; there is no shared vocabulary for data interchange; and the document model is unable to track structural changes.

Similarly, we have seen in section 2.2 that the employment of tree structures is equally incompatible with our complex document requirements. I have shown that the use of the tree model can lead to a series of problems. Overtagging and exponentially growing complexity necessarily leads to a loss of readability and interpretability. The limitations of the data model tempt philologists to workarounds, resulting in ambiguous data models and idiosyncratic documents. The workarounds then must be painstakingly counteracted in code, as workarounds affect all downstream tooling. Idiosyncratic data also is not interoperable across projects. Such practices are all-round unsustainable, making this document model equally unsuitable for long-term preservation.

From this perspective it is necessary to establish a new document model which goes beyond

and ideally encompasses both linear plain text and the hierarchically organised tree model. Following the syntax-model-validation paradigm, the document model I will propose in what follows will need to be accompanied by a suitable syntax, together with access and transformation methods, just as the current data models are.

The next chapter will present my design of such a data model for the interoperable long-term archiving of textual cultural heritage. The raison d'être of the new model is to guarantee meaningful long-term preservation and human-centric data curation.

Chapter 3

Design

This chapter introduces a new type of document model which natively supports concurrent text with concurrent annotation. Its name is *codex*.

codex's document model is more powerful than tree-based document models. It supports concurrency on the levels of text, annotation and hierarchy, and also in the dimension of time. As a full replacement to linear and tree-based document types, the design presented here includes a document model, together with mechanisms for access and mutation, as well as transformations to other data structures needed for further processing. The document model is also accompanied by a novel type of syntax, called the topological notation, for serialisation and long-term archiving.

3.1 Topological notation

I suggest a novel notation format, which I call topological notation, with the goal of enabling concurrent annotation over concurrent text. The topological notation is a printable format, but it is also the abstract document model. The format is inspired by music notation and requires only few key principles.

Fundamentally, it builds on a *topology*: all of its operations are defined in the two-dimensional space spanned by the specific layout of information in it. In the topology, symbols gain their meaning through their topological arrangement, which makes the majority of explicit markup unnecessary.

3.1.1 Description and Walkthrough

The topological notation represents a multi-dimensional matrix of interconnected and synchronised layers. It uses a tabular layout to display and linearise the data.

Listing 3.1 demonstrates the notation format, using the example from the introductory chapter. Some annotation is added for this example, namely linebreaks (layer `a2`) and the output of a part-of-speech tagger (layer `a1`) .

Listing 3.1 Topological notation

```
.   .             .           .   .   .       .    .        .            .       .   .       .       .     |
                                  let us fathom     vast                                                   | r3: b0
Such              ----------                                                                               | r2: r1
----              like these                                                                               | r1: b0
Such considerations           can show us       the infinite  possibilities of the means of our language  | b0
     -------------- NOUN          ---- VERB                   ------------- NOUN   ----- NOUN   -------- NOUN  | a1: b0
                                                                                      ^ <lb/>               | a2: b0
```

At its core, the format is made up of interdependent *layers*. Each layer is represented by its own row in a textual matrix. Layers can contain textual data, structural information, editorial operations by the original author, as well as any annotations philologists wish to add. Layers are identified by unique labels such as `r3`, shown here on the very right hand side. Note the dots on the top-most line. These points are synchronisation points. Layers consist of sections, which are the horizontal segments between synchronisation points. The synchronisation points thus serve to horizontally synchronise sections from different layers.

There is exactly one layer with first-class citizen status, and that is the *baseline* layer, marked `b0` in the above example. It contains the string "*Such considerations can show us the infinite possibilities of the means of our language*". This corresponds to the original sentence written by Wittgenstein, before textual edits occurred.

The baseline has a special status in that it separates the notational format into two tiers: textual layers representing editorial operations, which are placed above the baseline, and the annotation tier, containing annotation layers below it. This is a canonical ordering I have designed, although vertical ordering does not encode any specific semantics. If the layers appeared in non-canonical ordering, the interpretation of the data by *codex* would be identical, as long as the labels are correct[1].

Layers `r1` through `r3` on top of the baseline `b0` are textual layers. Textual layers record editorial operations in the original text; these can contain deletions indicated by "`-`" and insertions indicated by text snippets above empty space. Open variants are also expressable in this format. They correspond to textual layers where text snippets occur above (and aligned with) already existing text.

Layers `a1` and `a2` are annotation layers, which can be freely added by philologists. Layer `a1` shows the output of a part-of-speech tagger, which added information such as the respective part-of-speech of select lexical items (here: nouns and verbs). Layer `a2` indicates line breaks in the original text.

[1] In that respect, vertical ordering is different from the horizontal ordering, which centrally determines the semantics of the topological format.

There are dependencies between the layers, which form a hierarchy. For instance, layer `r1` builds directly on the baseline layer `b0`. This is expressed by the notation `r1: b0` at the very right-hand side. We call `b0` the *parent layer* of `r1`. Note that the base layer `b0` has no parent layer, indicating its fundamental nature in this construct, and leading to its first-class citizen status. In contrast, textual and annotation layers always have parent layers. Importantly, annotation layers can have *multiple parents*; this will help us to represent multiple hiearchies. Their purpose is to contribute textual revisions or annotation to specific sections of their parent layer.

Note that the editorial operations expressed by layer `r1` are coordinated, which records one philologist's interpretation of the text. Here, this is expressed by the fact that the coordinated editorial operations occur on the same line. Textual layers `r2` and `r3` also express coordinated editorial readings.

By placing text in a 2-dimensional matrix, similar to music notation, meaning emerges through a combination of the topology, synchronisation points, and the layer hierarchy.

I will now discuss how the topological format represents editorial operations.

3.1.2 A first glance at topological editorial operations

The topological arrangement has the ability to encode textual variants precisely and declaratively, without requiring explicit markup. In its usage of physical space, the notation is close to handwriting on paper and follows similar principles.

Three main patterns of topological arrangement can be identified for the three main editorial operations, namely those for *insertions*, *deletions* and for *variations*. Recall that variations are unusual textual arrangement used by some authors, including Wittgenstein.

The arrangement shown in listing 3.2 corresponds to a deletion. The symbols "`-`" are used to indicate the deleted material below it; the deleted text piece is therefore the word "*Such*".

Listing 3.2 Topological deletion arrangement

```
.   .                   .
----                    | del: base
Such considerations...  | base
```

The textual layer `del` that encodes this deletion is declared as being dependent on the baseline layer `base`, on which it operates, through the notation `del: base`. From a different perspective, one could say that layer `del` "applies" its change on its parent layer `base`.

The arrangement shown in listing 3.3 corresponds to an insertion, namely of the string "*like these*".

Listing 3.3 Topological insertion arrangement

```
.              .          .               .
               like these                 | ins: base
Considerations            can show us... | base
```

Note how additional space was inserted into all layers of the matrix, in order to make space for the inserted textual material. Again, this textual layer is a child of the baseline layer, as we can read off its label `ins: base`.

If one layer applies symbols onto the empty space of its parent layer, then it is interpreted as an insertion. This means that the semantics between the presence or absence of symbols is interpreted topologically.

Finally, Listing 3.4 shows an open variant, the third arrangement treated here. Open variants indicate an alternative, or possibly additional, second wording, resulting in a new kind of reading possibility. It involves a variation between "*show us*" and "*let us fathom*". The variant would result in two distinct readings. Note how these two text pieces are synchronised; additional space was inserted to pad the length difference between the two text pieces.

Listing 3.4 Topological variation arrangement

```
.                        .               .                            .
                         let us fathom                                | var1: b0
Such considerations can  show us        the infinite multitude... | b0
```

These three operations build the basic set of editorial operations. They can be combined recursively to represent more complex constructs, such as substitutions and transpositions, as will be shown in more detail in section 3.4.3 on compund editorial operations.

One caveat should be considered because we aim for a format that is friendly to the human eye. The topological notation creates very long matrices, which can be difficult to read if left as they are. In traditional texts, we have a similar problem in that very long lines would demand horizontal scrolling to be taken in. This is why traditional text editors usually wrap long lines at a certain position.

Codex offers an analogous `format --split` functionality, which allows us to split large blocks into smaller pieces, as well as functionality to re-join blocks back together (via the sub-command `format --join`). In the topological notation, extra care needs to be taken when performing this "wrap" operation, because multiple lines must be wrapped together as a block as opposed to mere line-wrapping.

This concludes our first glance a the topological format. The similarities between editorial operations applied by ink on paper, and those in the topological format, are summarised in Table 3.1.

	Handwriting	Topological notation
Insertion	Insertion text is written into empty physical space (often using a symbol like ⋁ to indicate its logical position)	Insertion text creates new topological space on parent layer
Deletion	Deleted material is crossed out	Delected material is crossed out using `---`-symbols
Variant	Variant text is placed between the lines, on top of variant base	Variant text is placed on a separate layer on top of the variant base; layers are horizontally aligned

Table 3.1: Usage of physical space in handwriting vs. topological space

3.1.3 Advantages of spatial organisation

The spatial organisation of using two physical dimensions in the topological notation offers a variety of advantages over embedded markup and the tree model.

- The topological notation format presents an intuitive, interoperable, declarative, human-intelligible and machine-readable format – properties that are crucial for long-term interoperable archiving.
- There exists a strict separation between primary text and any kind of additional metadata. (Note that I use the term "metadata" to mean any kind of annotation here, rather than data above the baseline.) Metadata is not directly embedded into the text through markup. Instead, metadata connects into the text from outside, in a manner akin to standoff annotation, using the horizontal synchronisation points to anchor the meta-data in the text. In my design, the standoff annotation resides in the same data structure (or file) as the annotated data, which insulates it from the risk of metadata getting lost.
- There is no necessity for the document structure to conform to a mono-hierarchical organisation. Rather, documents can represent any desired internal structure, be it linear, hierarchical or multi-hierarchical. This is made possible through two mechanisms: Firstly, the model allows to express both containment relations (the

parent/child dimension) as well as sibling relations (previous-child/next-child) for hierarchical structures. Secondly, metadata can be placed concurrently across separate annotation layers to make multi-hierarchical organisation possible. As a result, representing overlapping and non-contiguous structures no longer poses a problem.

Because annotation does not need to be forced into a mono-hierarchical tree structure, the need for workaround methods such as artificial fragmentation is eliminated. This also frees us from idiosyncratic procedural re-combinations of text pieces, otherwise necessary to reverse the fragmentation during data access.

- The data model can also be used for the recording of revisions to the document's structure itself. This is made possible because multiple non-linear base texts as well as concurrent hierarchies can be represented in parallel.

One major design feature of the human-centric approach described here is that it is geared at the human visual perception apparatus. One of the important principles is that annotation never obstructs the primary text, which always remains legible. The nature and composition of layers can be immediately taken in and their interpretation is obvious and natural.

Should misconfigurations or mismodelled pieces arise, then the human eye can easily spot them. In principle, this format could even be printed out on paper, which makes it particularly suited for the very-long-term archiving situation. And yet, the intuitiveness of the format does not stand in the way of its full machine-readability.

How was this possible? The most important conceptual aspect of the solution is the shifting up in dimensionality from one to two. This enables us to do away with embedded markup, with its troublesome matching of opening and closing tags within an ordered hierarchy. The same job can now be done by the topological arrangement of information with its vertical stacking and horizontal synchronisation. When shifting from one dimension to two, we also shift from a mono-hierarchical data model to a data model with synchronised, multi-dimensional concurrency.

The value of this work derives from a combination of two factors: simplicity and universality. Both the architecture and the data model are kept very simple in order to make data retention future-proof. If it ever becomes necessary in the future, the simplicity ensures that it should remain simple and self-explanatory to implement this structure in other languages or on novel computer architectures.

3.2 Concurrency in the topological document model

The shift to recording information through a topological organisation in two-dimensional space provides native support for concurrency. There are different kinds of concurrency in the DH scenario, which operate along different dimensions:

- Concurrency on the level of the primary text. This concerns non-linear text created by editorial operations. Concurrency on the level of text means that the document model should be able to represent alternative versions to the base text. The document model should ensure synchronisation with the original version.
- Concurrency on the level of metadata, annotation and hierarchies. This type of concurrency enables multiple hierarchies and overlapping annotation.
- Temporal concurrency. Parallel text is created by authors or annotators changing their mind, and by multiple authors working on the same document, either at the same time in a synchronous, collaborative context or in asynchronous long-term archiving and data curation tasks. In philology, we often want to capture changes over time and diverging opinions of experts.

I will now discuss these in turn.

3.2.1 Concurrent Text

As we have seen in the previous section, the editorial operation of the open variant leads to a multi-linear text, allowing for different reading possibilities. I will from now on refer to "reading possibilities" as linearisations. A linearisation is created by flattening a multi-linear text (for instance created by editorial operations), into a single linear text. The flattening process itself is also called linearisation.

Besides allowing philologists to check their work, linearisations also serve a different purpose. If we want to enable full-text indexing of the text for meaningful textual queries, the indexing has to be performed on the linearisations[2].

Listing 3.5 Variants without coordination

```
.                        .             .   .       .               .|
                         let us fathom                          [...] | v1: b0
                                           vast                 [...] | v2: b0
Such considerations can show us       the infinite multitude    [...] | b0
```

Listing 3.5 illustrates two independent variants, `v1` and `v2`. Notice how the long section "*can show us the infinite multitude*" is split into smaller sections (indicated by the top row of synchronisation points) through this addition of a variant operation.

[2]Full-text indexing could be performed using a symmetric index structure like SIS (Bruder 2012).

There are four possible linearisations of this construct:

1. Linearisation of baseline layer `b0`: *Such considerations can **show us** the **infinite** multitude of the means of our language*
2. Linearisation of baseline layer `b0` including layer `v1`: *Such considerations can **let us fathom** the **infinite** multitude of the means of our language*
3. Linearisation of baseline layer `b0` including layer `v2`: *Such considerations can **show us** the **vast** multitude of the means of our language*
4. Linearisation of baseline layer `b0` including layers `v1` and `v2`: *Such considerations can **let us fathom** the **vast** multitude of the means of our language*

The linearisations correspond to combinations of layers. They are read off the layers as follows:

- The first linearisation is produced by taking all sections from the baseline layer `b0`.
- The next linearisation is created by taking the baseline layer and applying the editorial operations of layer `v1` onto it. This is done by choosing layer `v1`'s respective section instead of the baseline's.
- The same happens for layer `v2`.
- The final linearisation is created by applying both changes to the baseline at the same time, i.e. accepting *let us fathom* from layer `v1` together with *vast* from layer `v2`.

In a scenario with n independent editorial operations one can derive 2^n linearisations.

However, the pure ability to record parallel versions is not sufficient to cover all multilinear cases; it must also be possible to declare *coordination* between variant readings. Whenever a reading is encoded as coordinated, this means that the annotator interpreted the operations involved as dependent on each other. Coordination allows a philologist to filter out unlikely readings.

Consider listing 3.6. Layer `r1` contains two editorial operations on the same line, which makes them coordinated. This means that these editorial operations contribute to the possible readings only in combination; they may not be interpreted independently.

Listing 3.6 Variants with coordination

```
.                          .             .   .       .               .|
                           let us fathom     vast              [...] | r1: b0
Such considerations can show us       the infinite multitude [...] | b0
```

From this coordination, the number of readings has dropped to two:

1. Linearisation of layer `b0`: *Such considerations can **show us** the **infinite** multitude of the means of our language*

2. Linearisation of layer `r1`: *Such considerations can **let us fathom** the **vast** multitude of the means of our language*

Practical Linearisation Example

In *codex*, linearisations are created by using the commandline API. I will now show an example sentence which contains not one, but two coordinated open variants (the reader may recall that this is the original sentence underlying our previous simplified example with one open variant).

Listing 3.7 demonstrates the resolution to *all* possible readings, using the `--all` flag, should this be desired.

Listing 3.7 Linearisation of all possible readings

```
> codex linearize --all such-considerations.codex
Such considerations can show us the infinite multitude...
Such considerations can show us the vast multitude...
Such considerations can let us fathom the vast multitude...
Such considerations can let us fathom the infinite multitude...

Considerations like these can show us the infinite multitude...
Considerations like these can show us the vast multitude...
Considerations like these can let us fathom the vast multitude...
Considerations like these can let us fathom the infinite multitude...

Considerations can show us the infinite multitude...
Considerations can show us the vast multitude...
Considerations can let us fathom the vast multitude...
Considerations can let us fathom the infinite multitude...

Such considerations like these can show us the infinite multitude...
Such considerations like these can show us the vast multitude...
Such considerations like these can let us fathom the vast multitude...
Such considerations like these can let us fathom the infinite multitude...
```

Listing 3.8 demonstrates the linearisation to the intended readings only, as opposed to all possible combinations, for the same sentence.

Listing 3.8 Linearisation of intended readings

```
> codex linearize such-considerations.codex
Such considerations can show us the infinite multitude...
Such considerations can let us fathom the vast multitude...
```

I will now explain how I build the relevant data structures that support the reading out of linearisations.

The Text Object Model (TOM) as a transducer Consider Figure 3.1, which shows a directed acyclic graph I call a variant graph, using our example from earlier. Linearisations correspond to traversals of the variant graph.

Figure 3.1: Variant graph (non-coordinated readings)

In this DAG, the states can be regarded as (abstract) positions in the text. They represent the horizontal synchronisation points of the topological notation; we could call it the "progress". The edges between the states carry the section information, both the textual content of this section as well as the section's layer identifier. If the edges do not carry such information, then any path is viable.

The graph structure in Figure 3.1 does not have any labels on its edges. This corresponds to the un-coordinated scenario above. Accordingly, four different paths through this graph are possible, corresponding to four linearisations.

The DAG in Figure 3.2 is different. This graph carries labels on its edges, restricting the possible ways in which this graph can be traversed. This scenario corresponds to the coordinated open variants seen above.

Figure 3.2: Variant graph (coordinated readings)

The variant graphs used in *codex* are called the Text Object Model (TOM). TOM traversal works as follows: We start at the start state 0, with a particular layer in mind (here: `r1`). (Note that the start state has no incoming edges). The algorithm now checks the outgoing edges whether we have a valid ticket for traversal. If there exists an edge for which we have a ticket, then this edge is chosen and we proceed to the next state. Upon transition, the textual content of the chosen edge is emitted. In our case, we have tickets `r1` and `b0`. (I will explain how we got to be in possession of these tickets in a second.) This process is repeated until there are no more valid edges to choose from. If needed, the algorithm backtracks to a state where other choices of transitioning would have been possible and continues from there. The algorithm makes sure that once one has made a commitment to one layer, then only valid paths for the linearisation corresponding to the layer can be followed. In the DAG above, this algorithm results in the desired two linearisations:

Such considerations can ***show us*** *the* ***infinite*** *multitude of the means of our language* (`b0`)

Such considerations can ***let us fathom*** *the* ***vast*** *multitude of the means of our language* (`r1`)

No other traversal possibilities exist, and this is as it should be.

There is now only one puzzle piece missing in the solution, and that is where the tickets come from. The TOM doesn't act alone; I have designed a team of graphs that act as the joint data structures for creating linearisations. The two graph structures are co-joined and work in close collaboration, hence the expression "team of graphs". The second graph is a Document Object Model (DOM)[3], which tracks the dependencies between layers. Through the DOM-TOM interaction, variant graphs similar to those by D. Schmidt and Colomb (2009) can be built, which in turn allow for the linearisation of the reading variants. The purpose of the DOM is to provide the pieces of information that are necessary to walk the TOM and read off linearisations. Specifically, the DOM provides the tickets for the traversal of TOM's edges.

Document Object Model (DOM) as a layer tracker The DOM is essentially a model of the dependencies between layers. In the DOM, the nodes correspond to layer identifiers, and edges express the relation between layers, corresponding to "is dependent on". Effectively, this creates an *inheritance chain* between the layers.

Dependencies between layers in my framework work as follows: Layers with editorial operations *mutate* their parent layer; this means that *a)* the layer applies its own textual changes to it, and *b)* a new layer is created. Layers can only apply their changes to their parent layer. For instance, the textual layer with id `i1` can only apply its editorial operations to its parent layer `b0`; this dependency is expressed by the identifier marks `i1: b0` on the very right hand side of the topological notation. Each newly created layer can then be recursively modified by other layers. Figure 3.3 gives an example DOM.

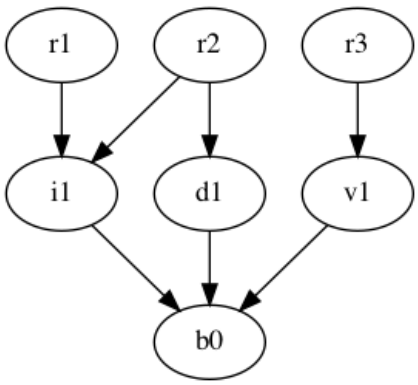


Figure 3.3: An example DOM

In this example, three editorial operations are applied to the baseline `b0`: layer `i1`, an insertion; the layer `d1`, a deletion, and finally layer `v1`, an undecided variant. Layer `r2`, as

[3]Note that DOM is a term that is also in use in the XML world, but with entirely different meaning. In XML, a DOM is a tree structure, whereas here, a DOM is a directed acyclic graph structure.

an annotation layer, can have more than one parent; here, it has two parent layers (`i1` and `d1`). A constellation like this would mean that layer `r2` can be applied to either layer `i1` or `d1`. (Recall that readings in this context are combinations of layers.)

Let us now see how the DOM tracks these dependencies and creates tickets for the TOM. If we want to create the linearisation for a layer, say `r1`, then we start from the node corresponding to this layer. From here we follow all possible paths along the inheritance chain of layers until there are no more edges to follow (we will have landed at the baseline `b0` by then), and collect all labels from the states that we pass, into a set of tickets. For instance, when linearising layer `r1`, the set of tickets would be {`r1`, `i1` and `b0`}. This set of tickets is now available for our guided traversal through the TOM.

3.2.2 Concurrent Metadata and Annotation

This section discusses annotation layers. The topological format models annotation using pen and paper as closely as possible. Annotation in handwriting is often handled by underlining a certain piece, and placing an annotational remark close to it. The topological format proceeds analogously.

In general, annotation layers follow the same conventions as textual layers: they connect to the text through topological arrangement; they carry an identifier as well as an identifier to their parent layer they have a dependency to. An example for how annotation works can be found in listing 3.9. Like in handwriting, the area to be annotated is 'underlined' and joined by a small remark.

Listing 3.9 Simple Annotation

```
> cat such-considerations.codex |\
  codex add-annotation --id a1 --pid b0 --at  5 --len 14 --data "NOUN" |\
  codex add-annotation --id a1 --pid b0 --at 63 --len 9  --data "NOUN" |\
  tee such-considerations.codex
.   .              .           .   .              .   .        .               .
----                                                                 [...] | del1: b0
                    like these                                       [...] | ins1: b0
                                    let us fathom     vast           [...] | var1: b0
Such considerations             can show us       the infinite multitude [...] | b0
     -------------- NOUN                                    --------- NOUN  | a1: b0
```

Listing 3.9 shows a simple form of annotation. What we have here is a valid document, which is topologically expressed, with minimal means.

Note that the annotation layer `a1` annotates two regions of the textual layer `b0` as `NOUN`. Annotation is polyglot and can represent anything one might want to express. Multiple annotation schemes can be mixed. This is advantageous, because one might wish to describe one and the same text as belonging to different markup schemes. A document could be

annotated with markdown, LaTeX, OpenDocument XML as well as TEI, simultaneously. *codex* as a polyglot document model is able to represent them all in parallel with no trouble.

Partial annotation means that we can casually annotate whatever we feel like without the burden of having to establish a full hierarchy (as we would if we used XML). In order to annotate this little piece, in the XML case, we would need to introduce a root element and potentially several other elements (depending on the schema). Only then would we be able to use the NOUN element to annotate these text snippets. *codex* is lean. In *codex*, we pull in information if desired, rather than walking down a tree until we reach the desired information.

Listing 3.10 Pointy annotation

```
> cat such-considerations.codex |\
  codex add-annotation --id a2 --pid b0 --at 32 --len 1 --data "<lb/>" |\
  codex add-annotation --id a2 --pid b0 --at 63 --len 1 --data "<pb/>" |\
  tee such-considerations.codex
.   .               .          .    .             .   .       .         .      .
----                                                                    [...] | del1: b0
                    like these                                          [...] | ins1: b0
                                       let us fathom     vast           [...] | var1: b0
Such considerations             can show us       the infinite multitude [...] | b0
     -------------- NOUN                                        --------- NOUN  | a1: b0
                                  ^ <lb/>                       ^ <pb/>         | a2: b0
```

Listing 3.10 shows an annotation that does not cover a range, but instead annotates a point. Such annotation is referred to as "pointy annotation" and in that respect is similar to empty elements in XML. Since the pointy annotations exemplified here use XML/TEI's `<lb/>` and `<pb/>` tags, to indicate a line break and a paragraph break, this example also demonstrates the possibility to mix multiple markup schemes.

Listing 3.11 Annotation of arbitrary textual layers

```
> cat such-considerations.codex |\
  codex add-annotation --id a3 --pid ins1 --at 5 --len 19 --data "pencil" |\
  tee such-considerations.codex
.   .               .          .    .             .   .       .         .      .
----                                                                    [...] | del1: b0
                    like these                                          [...] | ins1: b0
                                       let us fathom     vast           [...] | var1: b0
Such considerations             can show us       the infinite multitude [...] | b0
     -------------- NOUN                                        --------- NOUN  | a1: b0
                                  ^ <lb/>                       ^ <pb/>         | a2: b0
                    ---------- pencil                                          | a3: ins1
```

Listing 3.11 shows many of the annotation types we have encountered up to now, in one example. Specifically, the newly inserted annotation layer `a3` annotates the insertions of *"like these"* to have been made using a pencil.

Listing 3.12 Overlapping annotation

```
> codex add-baseline --data "Lorem ipsum dolor sit amet" --id b0 |\
  codex add-annotation --data "bold" --at 6 --len 5 --id a1 --pid b0 |\
  codex add-annotation --data "italics" --at 9 --len 8 --id a1 --pid b0
.     . .  .     .        .
Lorem ipsum dolor sit amet | b0
      ----- bold           | a1: b0
         -------- italic   | a2: b0
```

Listing 3.12 demonstrates a case of overlapping annotation. In contrast to overlapping annotation and necessary workarounds in the tree model (cf. section 2.2.1 on page 31), this annotation remains simple and straightforward and no workaround methods are necessary to encode this kind of information.

Thus far, I have demonstrated that *codex* can encode overlapping as well as multi-hierarchical annotation which have created severe problems in the tree-based model. Despite this complexity in the annotation, the visual representation has remained satisfyingly simple.

In sum, we have seen here that *codex* can encode full document structure or only partial annotation information. Annotation can also combine multiple document markup schemes at once. Pointy annotation is supported.

Let us now move on to the next kind of concurrency.

3.2.3 Concurrent Hierarchies and Document Structures

Concurrent hierarchies stand in close relation to concurrent metadata. In fact, it can be seen as a special case of concurrent metadata.

This type of concurrency can surface in a variety of scenarios:

- **Structural changes**. Revision of the document's structure leads to concurrent hierarhices. For instance, the "Track Changes" functionality in word processors requires massive amounts of workarounds to make a paragraph join possible, as we have seen in section 2.2.3. In *codex* this is trivial.
- **Multi-dimensional descriptions**. Linguistic structures such as words, sentences and paragraphs are often not aligned with the material structure of page dimensions and lines lengths. This requires a multiple hierarchy, which can only be expressed with workarounds in the tree model. In *codex* this scenario again is trivial.

For both scenarios we need one or more hierarchies. In my design, annotation can not only refer to the textual parts of the topological notation but also to other annotation layers.

This latter fact can now be utilised to make fully hierarchical or concurrent hierarchies possible, as will be shown in the following practical examples.

Practical Example: Building Concurrent Hierarchies

I have earlier talked about *codex*' ability to represent partial hierarchies and have argued how practical this is. However, *codex* can of course also represent full hierarchies. Listing 3.13 shows one such single hierarchy. This full hierarchy is equivalent to a simple XML document. First, annotation layer `a1` covers the whole document. This is the equivalent to the root node in XML. Then, there are multiple annotation layers recursively referring to their respective parent layer. Note how all child-layers are fully contained within their parent layer and no layer overlaps with another layer.

Listing 3.13 Full hierarchy in the topological format

```
Lorem ipsum dolor sit amet             | b0
-------------------------- <document>  | a1: b0
---------- -------------- <paragraph> | a2: a1
----- ----- ----- --- ---- <word>      | a3: a2
```

Listing 3.14 adds a concurrent hierarchy by describing the material dimensions of the manuscript page.

Listing 3.14 Concurrent hierarchy in the topological format

```
Lorem ipsum dolor sit amet             | b0
-------------------------- <document>  | a1: b0
---------- -------------- <paragraph> | a2: a1
----- ----- ----- --- ---- <word>      | a3: a2
-------------------------- dim:page    | d1: b0
---------- -------------- <paragraph> | a2: a1
---------- -------------- dim:line    | d2: d1
```

Here, two more annotation layers are added, describing the material dimensions of the manuscript page: layer `d1` describes the page, whereas layer `d2` describes the lines written on that page. Note how layer `d2` actually overlaps with the paragraph annotation of layer `a2` and also, how multiple markup schemes are put to use: while layers `a1` through `a3` use xml, layers `d1` and `d2`, in this case use RDF (which can be assumed to have been imported elsewhere).

In sum, *codex* is able to represent one or more hierarchies in a visually simple and logical way and can thus avoid many problems of other document models. It doesn't have to resort to artificial fragmentation or other workaround methods which might potentially be harmful.

3.3 Validation in the topological document model

Many of the constructions that are considered illegal in XML are purposefully allowed in codex. In XML we need machinery to check that these constraints aren't violated. Here, they cannot possibly go wrong so no checking mechanisms are necessary:

- Overlapping markup is no problem and is fully allowed.
- Multiple hierarchies are represented in separate layers and are thus not a problem.
- Multiple markup schemes in parallel are not a problem.

However, there is a big difference between codex and XML: codex is like a blank sheet of paper. The philosophy of codex as a polyglot format is that it is agnostic towards the exact nature of the annotation schemes that are used in it. Rather, it allows for the use of multiple markup schemes in one and the same source. However, using the right type of annotation, these differently marked-up sections can programmatically be taken apart again and be exported to the respective format behind it. It intentionally leaves the task of keeping consistency up to the user. While not being directly involved with any of the annotation schemes, it can still provide support to a user to enable them to get as little as possible wrong, at least amongst the things that codex has information about.

The ideas of well-formedness and validity, which we know from XML, therefore have corresponding concepts in the topological format.

- Well-formedness is now corresponding to some basic properties such that blocks are compatible. This can be realised through some **consistency checks**.
- Validity concerns semantics and intended structure. It is of course harder to maintain and to check. In codex, the user is supported in checking these herself.

3.3.1 Consistency

Given how permissible codex is, we need to ask ourselves what could even go wrong with its syntax. Note that most of the errors I will describe now are only possible when a user directly interacts with the topological format, i.e., edits the files by hand. These things should not be possible when the APIs (such as the commandline API) are used programmatically, or when a graphical interface is used.

- If the user uses non-allowed symbols (e.g. "`++++`" instead of "`----`"), codex would interpret this as text, i.e., as an open variant. Some checks could be built in to avoid obvious wrong cases.

- If the user has forgotten to add a layer identifier, the layer is not represented as a node in the DOM, with all follow-on errors.
- A similar inconsistency in the DOM could be caused by the user forgetting to specify the parent id of a layer; then both would wrongly be interpreted as baselines.
- If the user specifies the parent of a node as the node itself, the DOM would no longer be an acyclic graph.

All DOM-related errors are also easy to spot and automatically detected by the parser. In any case, these types of errors are already refused on the API-level to make such cases impossible.

3.3.2 Validity of the data model

What I understand as validation is this: the philologist wants to check whether the textual processes that she has modelled actually lead to the intended results. These checks are supported by the variant graphs. The philologist might find during this type of validation that what she has modelled does not correspond to the intended results. If that is so, she has to make changes to her model, such as different dependencies between the layers (and the resultant change in the hierarchical relations and names of layers). This will result in different linearisations, hopefully in the correct ones. Maybe what she has to do is to model the sections differently. The system should provide support for such semantic checks. If there was a graphical user interface, it could run these operations constantly in the background, displaying them upon request.

3.3.3 Correctness in alignment of sections and layers

Validation and consistency checks could be specialised for other kinds of errors occurring in the topological format too. For instance, even if a record once was syntactically and semantically perfect, it might not stay that way. Physical storage is not perfect, and some form of data corruption might dislocate sections, or destroy or move some synchronisation points. This could lead to a situation where sections aren't aligned with other sections or where entire layers are located in the wrong place.

One could think of adding some additional measures that would help re-align sections in such a case. For instance, one could think of adding special control markers from the very low region of `ASCII`, such as `\0` or `^V` (`ASCII` code `22`, named "`SYNCHRONOUS IDLE`") to every layer at specific distances. These special control characters do not usually get displayed unless special measures are taken, but would still be there to help us ensure correct alignment. One might even think of how one could economise on such control

markers, by placing more of them at the beginning of a record, and fewer of them, in growing distances, later. The Fibonacci sequence might provide the distances. If control sequences at the beginning align, we could be increasingly certain that all parts are in the correct places (up to the current control character). Should sections be dislocated, these distances can also give us good indicators of the degree of distortion.

3.3.4 Validation of foreign formats

Codex also gives the user the freedom to export the data to external formats where there are established tools that can check for validity in these respective worlds, if this is possible. Of course, this is not always possible, and we of course know that workarounds short-circuit any supposed "validation" of a document.

If data happens to be compliant with a meaningful XML schema, *codex* allows for smooth export into XML, so that the validation mechanisms of the XML world can be applied. Even if it doesn't, *codex* could provide another kind of consistency check in situations where users plan to export data to an external format such as XML. In that case, *codex* could check for things it knows that could cause trouble in that format, such as overlapping or non-contiguous markup. *Codex* could provide assistance in such situations, even beyond the obvious warning to users that their data is in contradiction with existing output formats. In the extreme case, if for whatever reason some non-XML-compliant data *h*as to be exported into XML and workarounds are therefore unavoidable, *codex* could perform the fragmentation method programmatically, at least ensuring consistency and correct labelling of the fragmented parts.

3.4 Textual revision in the topological document model

The sort of textual concurrency we are after is not directly supported in computer science. Two basic operations, namely insertion and deletion, are known and treated in CS; to my knowledge, nobody in traditional CS has ever attempted to define editorial operations going beyond those two. Why only two, and why those two? The model of textual revision commonly used in computer science practice, for instance in version control, doesn't support textual parallelism. Textual models in computer science are always strictly linear; you cannot write between the lines on an electronic device. This assumption is hard-wired into systems such as `git`. From this viewpoint, the two editorial operations of deletion and insertion suffice to represent everything that is needed. In order to keep texts consistent under this model, textual conflicts always have to be resolved to a single textual version, ideally as quickly as possible.

We have however already seen one editorial scenario encountered in philological work, the *open variant*, that cannot be adequately described with the instruments given, so something more than those two operations is needed.

At the same time, philologists in practice use a set of editorial operations that goes beyond deletion and insertion, without ever having developed a formal theory of editorial operations or even just a formal definition of a definite set of editorial operations. Instead, every project seems to use whatever operations they think are useful. Despite this, when discussing this question with many practitioners in DH, I found that there is a general convergence to a set of roughly a handful of such editorial operations.

In an ideal world, there would be a philologically-inspired formal theory of editorial operations; this would be central to the interoperable recording and archiving of revisions of text. Beyond that, such a theory could also be useful in the context of collaborative editing environments such as Google Docs, where two or more users work together on a document and synchronously make editing changes. The current section lists principles for general editorial operations, in the hope that both fields could equally benefit.

To my knowledge, what follows is the first formal description of a fixed set of editorial operations.

3.4.1 A new model of textual revision for concurrent text

Classically, an insertion operation takes a piece of text, adds new text to it at a specific position, and thereby leads to an new passage of text. Similarly, a deletion, as the inverse of the insertion, takes a piece of text, deletes parts of it and yields a new passage.

Interpreting these operations from our matrix-like view we find the following:

- an editorial operation either carries textual *data* (as in the case of an insertion) or does not carry data (deletion)
- an editorial operation is carried out over an *extension* of the baseline, either a point or a range. In the case of the insertion, which happens at a specific point in the document, the extension is 0. While the insertion's data has a length, it is positioned at a specific place with no extension. Notably, insertions happen at the space *between* two characters. In the case of a deletion, the extension is the entire length of the deletion.

From this, we can see that the open variant is similar to an insertion in that it adds new text (and yields a new reading), but is also similar to the deletion in that it also has an extension. Instead of applying itself at a specific point (like the insertion does), it extends over a region of the base text.

Thus, the underlying logic of the three basic relations is as follows:

- text $\rightarrow$ space $\equiv$ insertion (When text is applied to previously unoccupied space, it is understood as an insertion and new topological space is created.)
- space $\rightarrow$ text $\equiv$ deletion (The application of space on a given text is understood as a deletion: the previously given text is effectively discarded and existing topological space is reclaimed.)
- text $\rightarrow$ text $\equiv$ variation (When a second piece of text is placed in the same place where there already was a piece of text, both are understood as variants of each other and topological space is adjusted.)

From a topological perspective, both an insertion and a variation create (new) topological space, whereas a deletion frees up topological space. Similarly, like the $\bigvee$ symbol in handwriting suggests, the inserted text extends the topological space at a specific point.

When an author creates an open variant, they add a second or third word in parallel with another word, without crossing out the former. From the modelling perspective, open variants can therefore be interpreted as an insertion with an extension.[4]

In what follows, I will first describe the three basic editorial operations, followed by the compound operations based on combinations of the former.

All operations will be introduced by *a)* giving an intuitive description of the operation, *b)* an example of its representation in topological notation, and *c)* the resulting linearisations that can be derived from them.

3.4.2 Basic Editorial Operations

Three basic editorial operations are given by the operations of *insertion*, *deletion*, and *open variant*. While the former two follow the intuitive concepts associated with them, as well as the diff-model of textual revision, the open variant is an addition.

Insertion

An insertion follows the regular concept from the diff-model, where text is added to already existing text. In the very beginning of a document, there is one insertion where text is added to empty space, namely the baseline.

In topological terms, an insertion extends the existing topology by creating new topological space. The resulting topological space is empty and represented through blank space. The

[4]Along these lines, a deletion could actually be interpreted as an *insertion*, namely an insertion of null-space in the topology.

semantics of the insertion are topologically clear, with the insertion exactly fitting into that space.

Listing 3.15 Insertion Matrix

```
.           .     .         . |
             dolor             | ins: base
lorem ipsum       sit amet    | base
```

After the insertion, there are two resulting linearisations. Listing 3.16 lists these, together with their provenance (indicated by the `[layer labels]`).

Listing 3.16 Insertion linearisations

```
lorem ipsum dolor sit amet  [ins, base]
lorem ipsum sit amet        [base]
```

Deletion

On the textual level, a deletion operation matches the intuitive concept of elimination of a piece of text.

From a topological perspective, it reclaims previously taken-up topological space. A deletion thus represents the inverse of an insertion. Listing 3.17 shows the deletion in topological notation. A deletion is carried out in topological notation by explicitly marking a certain region of a certain layer as deleted by inserting the respective number of `-`-signs.

Listing 3.17 Deletion Operation

```
.           .     .         . |
             -----             | del: base
lorem ipsum dolor sit amet    | base
```

Listing 3.18 lists the expected linearisations, together with their provenance.

Listing 3.18 Deletion Operation Linearisations

```
lorem ipsum sit amet        [del, base]
lorem ipsum dolor sit amet  [base]
```

Variant

The open variant is not covered by the classic diff-model of editorial operations. The open variant operation adds a new, variant reading to an already existing base text.

Listing 3.19 shows the variant operation in topological notation. Two cases need to be considered.

In case the variant text is longer than the underlying text, the underlying text is extended to fill the space until the next synchronisation point (Listing 3.20).

Listing 3.19 Variation

```
.           .    .        . |
            color          | var: base
lorem ipsum dolor sit amet | base
```

Listing 3.20 Long Open Variant

```
.           .      .        . |
            colorem           | var: base
lorem ipsum dolor   sit amet  | base
```

Listing 3.21 Short Open Variant

```
.           .      .        . |
            color             | var: base
lorem ipsum dolorem sit amet  | base
```

Listing 3.21 shows the opposite scenario, where the base text is longer than its variant.

Listing 3.22 gives the expected linearisations, together with their provenance.

Listing 3.22 Variation Operation Linearisations

```
lorem ipsum color sit amet  [var, base]
lorem ipsum dolor sit amet  [base]
```

Practical Example: Creating records through the command line interface

Listings 3.23, 3.24 and 3.25 demonstrate the basic API functionality of creating baselines, adding deletions and inserting text on the command line. The command line input is given in blue at the top; the resulting record is shown below.

Initially, a baseline is created using the `codex` sub-command `add-baseline` (listing 3.23).

Listing 3.23 Baseline Creation

```
> codex add-baseline --id b0 --data "Such considerations can show us ..."

.                                  .
Such considerations can show us ... | b0
```

The parameter `--id` allows the user to specify a label, whereas the parameter `--data` expects the textual input in quotes.

Next, a deletion is added on the baseline using the sub-command `add-deletion` (listing 3.24).

We need to supply a new name for the deletion layer this will create (parameter `--id`) and an identifier of the parent layer of the planned deletion (parameter `--pid`). We also need to specify the beginning of the deletion as a character offset (`--at`) and the length of the deletion in characters (`--len`).

Listing 3.24 Basic API functionality: Deletion

```
> codex add-deletion --id del1 --pid b0 --at 0 --len 4

.   .                               .
----                                | del1: b0
Such considerations can show us ...| b0
```

Finally, an insertion is added on the baseline using the subcommand `add-insertion` (listing 3.25).

Listing 3.25 Basic API functionality: Insertion

```
> codex add-insertion --at 19 --id ins1 --pid b0 --data " like these "

.   .              .           .                .
----                                            | del1: b0
                    like these                  | ins1: b0
Such considerations             can show us ... | b0
```

Here, the parameters are as above, except that no length is given, instead the inserted text is supplied `--data`. Note that the user does not have to worry about relative length of the insertion; codex will simply create sufficient space in the underlying topological format.

3.4.3 Compound Editorial Operations

This section describes the set of compound editorial operations. Compound editorial operations are built from specific combinations of basic editorial operations, where, for instance, a substitution can be decomposed into the basic operations of a deletion, followed by an insertion.

Substitution

As in handwriting, in the substitution operation, a piece of text is deleted, and then a different piece of text is inserted into the deleted text's place.

The only distinction between a substitution operation and an open variant is that the open variant doesn't include a deletion. Yet, the deletion operation which is part of the substitution operation can be considered transitional in the sense that it doesn't contribute to the resulting possible readings of the entire text (cf. listing 3.27 below). To make the transitional nature of the deletion layer explicit, it is marked with a $*$-symbol at the end of its label.

Listing 3.26 Substitution

```
.           .   .           .
            dolor           | sub : sub*
            -----           | sub*: base
lorem ipsum color sit amet  | base
```

Listing 3.27 shows the resulting linearisations of the substitution operation. Note how the deletion layer is not linearised, because of its transitional nature. Note also how the canonical ordering of the format places the insertion on top of the deletion layer (listing 3.26).

Listing 3.27 Substitution Operation Linearisations

```
lorem ipsum dolor sit amet  [sub, base]
lorem ipsum color sit amet  [base]
```

Transposition

The editorial operation of a transposition follows its intuitive concept, where two pieces of text are deleted and exchanged for another. Effectively, these two pieces of text switch places. In this model, the editorial operation of transposition is modelled as two coordinated substitutions, where two pieces of text are substituted with the exact textual content of each other (listing 3.28). As with the substitution, a transitional deletion-layer is necessary but does not become part of the resulting linearisations.

Listing 3.28 Transposition

```
.           .       .           .
sit amet            lorem ipsum | tr : tr*
-----------         ----------- | tr*: b0
lorem ipsum dolor sit amet      | b0
```

Listing 3.29 lists the expected linearisations of this transposition operation:

Listing 3.29 Transposition Operation Linearisations

```
sit amet dolor lorem ipsum  [tr, base]
lorem ipsum dolor sit amet  [base]
```

Re-Instantiation

Lastly, the editorial operation of re-instantiation is introduced. Re-instantiation describes a case where initially, a text is deleted, but then this deletion is revoked again. A good example of this can be seen in the introductory example (figure 1.1 on page 15): Here, Wittgenstein originally deleted the very first word of the sentence ("*Solche*") only to revoke this deletion afterwards (indicated by the small dots below "*Solche*").

There exist two possible ways to model and interpret a situation like this: either it is interpreted as two consecutive deletions, where the second deletion deletes the former, or alternatively, it is interpreted as a deletion followed by an insertion, where the inserted text exactly matches the the previously deleted text. Note how in this latter case, the editorial operation of the re-instantiation actually is a self-substitution. My design leaves open the question which of these representations is the better one; this decision can be left up to how a philologist wants to use the model.

Deletion without continuation

For completeness' sake, there is one special form of an editorial operation which I don't cover in my current system: deletion without continuation. It is an obscure phenomenon, little known even in DH circles. Michael Nedo however modelled it in the Wiener Ausgabe from the beginning, since it is frequent in Wittgenstein's writing.

The deletion without continuation is a scenario where the author writes a sentence and, in the middle of it, decides to abandon the train of thought, deletes the last few words and continues to write the sentence in a different direction than the former was heading. In this sense, the original thought with which the sentence was started is discontinued; the sentence itself is continued, albeit in a different sense.

This rare phenomenon is not modelled in *codex* at the moment, but it should be modellable with my system with a few small tweaks here and there. The baseline that is abandoned is filled with deletion signs to the very end and another layer sitting on top becomes the new baseline (the 'transposed' baseline). This means, one could create a deletion that always covers the entire region to the end, in combination with the creation of a new baseline. This baseline would need an extra marker to be covered/discovered as such.

This concludes my formal list of editorial operations in *codex*. How exhaustive is the set of editorial operations that I have described here? I can say, from the many conversations I have had with Michael Nedo and many other philologists, this list should be rather exhaustive for the common cases. After all, no other editorial operations are modelled in the TEI Guidelines.

But then again, people can get pretty creative with pen and paper, and there is no guarantee that we have experienced all that there is possible.

In any case, this set of editorial operations substantially improves the situation with respect to which phenomena observed in the philological work can be modelled computationally.

3.5 Spatial region indexing in the topological document model

I will now turn to the question of how we can perform efficient structural search and data transformation in the topological document model. For this, I will borrow an approach from the area of theoretical computer science and information retrieval.

The *Region Algebra* invented by Miller (2002) lends itself particularly well to the topological notation of synchronised sections. Region Algebra allows the manipulation of ranges over

text regions, so that various operations can be performed. His main insight is that all annotation can be seen as a range over text in two-dimensional space. As a result, one can query for all regions that come before a given a region, for regions that overlap with a given region, for those that end in the same point as a given region, and so on. This holds whether the annotation is encoded as embedded markup or by a standoff mechanism.

In order to use Miller's approach for query and extraction of sub-models of the format, all that is needed is a topological notation and a suitable index structure to handle the respective regions of the format. With this approach, it also becomes possible to handle overlapping regions and hierarchical structures elegantly, since both my topological notation and Miller's Algebra natively support this arrangement. These properties taken together makes Miller's Region algebra an attractive choice for my problem.

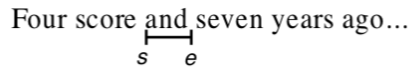
The next section will introduce the main ideas of Miller's work, followed by a description of its use for search, indexing and transmutation in my topological notation format.

3.5.1 Miller's Region Algebra

Miller's formulation of a Region Algebra (Miller 2002) presents an elegant approach to search and transformation of text with structured markup.

Encoding regions in two-dimensional space

Miller's main contribution is the idea to model document structure geometrically in two-dimensional space. Consider the following traditional one-dimensional representation of an annotation.



Miller proposes a 2-dimensional representation, where the x-axis corresponds to start positions of regions (s in the above example), and the y-axis to end positions (e in the example). A region, which has a horizontal spatial extension in one-dimensional space, now can be represented as a point, an object without any extension, as shown in figure 3.4.

Note that points in the lower 45° diagonal represent impossible regions (those that end before they begin) and can therefore be immediately discarded.

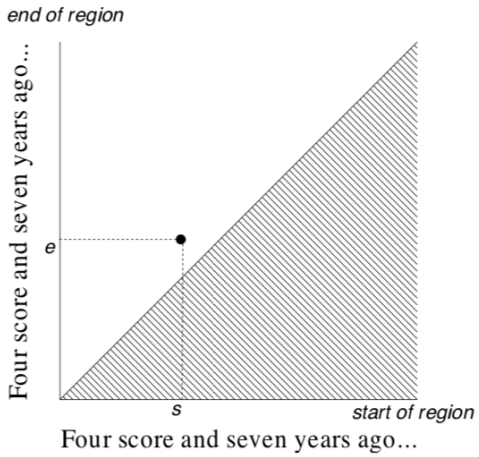


Figure 3.4: Points in region space (from Miller 2002)

Miller next establishes six geometric relations between regions (figure 3.5):

- before b/ after b
- contains b / in b
- overlaps-start b / overlaps-end b

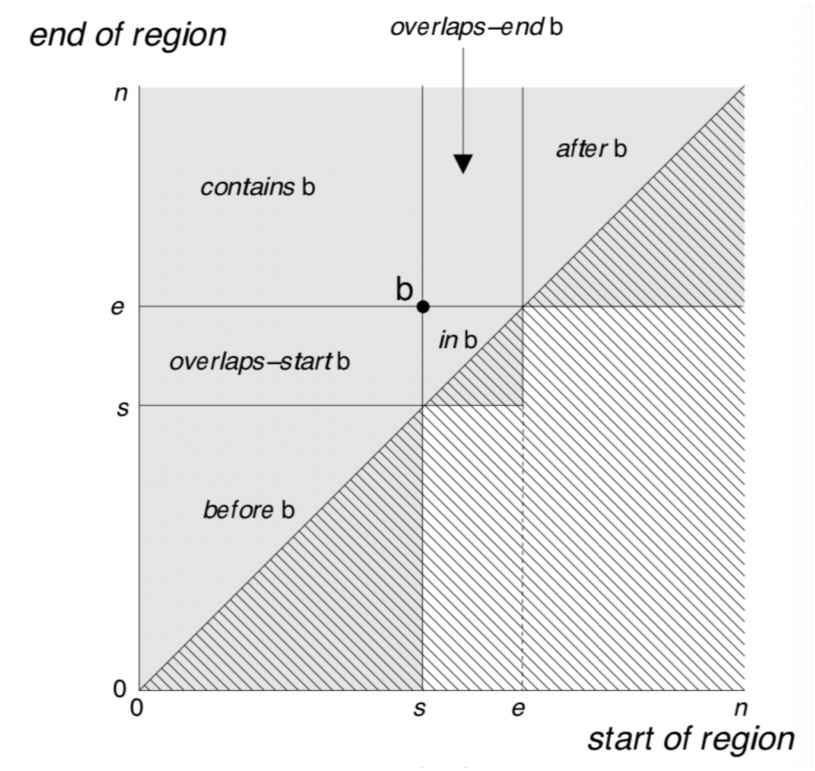


Figure 3.5: Fundamental Region Relations in two-dimensional space (from Miller 2002)

Each of these relations then maps to a corresponding field in two-dimensional space, defined in comparison to a reference point b (corresponding to a reference region in one-dimensional space). Every point (i.e. region) that falls into one of these six fields is then guaranteed

to stand in the corresponding relation to the reference region b. This is illustrated in figure 3.5; figure 3.6 shows the six relations in corresponding one-dimensional space.

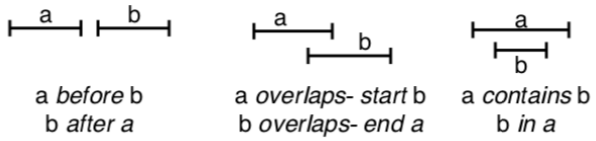


Figure 3.6: Fundamental Region Relations in one-dimensional space (from Miller 2002)

The definition of the relations between two text regions match our intuitions, as follows:

A before B The relation *a before b* represents a region a that appears in its entirety before b, meaning that it not only begins but also ends somewhere before b. Analogous reasoning applies to the opposite relation *after*.

A contains B The relation *a contains b* represents a region a that fully contains region b. Thus, region b will be smaller than a and its begin and end positions both appear within the range of a. Analogous reasoning applies to the opposite relation *in*.

A overlaps-start B The relation *a overlaps-start b* represents a region a that overlaps the start of region b, meaning that region a begins somewhere before b and ends somewhere in the 'middle' of b, i.e. somewhere before b's end, but after b's begin, thus overlapping b's start.

A overlaps-end B The relation *a overlaps-end b* represents a region a that begins somewhere in the middle of region b, and ends after b, thus overlapping b's end.

The four relations *begin/after* and *contains/in* stand in close relation to the basic relations of an ordered hierarchical tree structure that we have encountered before. The two relations *contains* and *in* correspond to the dominance relation between parent and child nodes, which operates vertically in a tree. The relations *before* and *after* correspond to the order between sibling nodes, which operates horizontally in a tree.

The final two relations, *overlaps-start* and *overlaps-end*, will enable the access to data in overlapping structures and structures with multiple hierarchies.

Using rectangles as query predicates

We have seen that every point that falls into one of these six fields is in the corresponding relation to the reference region b. Miller next introduces *rectangles* as a further abstraction of the point concept. A rectangle corresponds to the *sets* of regions that fall into one of these fields mentioned above (and thus are in a relation to the reference point). Rectangles

can be specified compactly, as only two coordinates are needed (the choice taken here is those of the lower left and upper right corners; points as defined in figure 3.5):

$$\begin{array}{ll} \text{before b} = (0,0,s,s) & \text{after b} = (e,e,n,n) \\ \text{contains b} = (0,e,s,n) & \text{in b} = (s,s,e,e) \\ \text{overlap-start b} = (0,e,s,n) & \text{overlap-end b} = (s,e,e,n) \end{array}$$

In this space, the biggest possible rectangle $((0,0),(n,n))$ can represent each and every available region. Regions (i.e. points) themselves can also be represented using this abstraction. This is done by constructing a trivial rectangle whose corners happen to coincide.

For rectangles, Miller's Region Algebra is closed under the set operations of union, intersection and difference. Starting from the six basic relations *begin*, *after*, *contains*, *in*, *overlaps-start*, and *overlaps-end* together with set logic with its associated operators, more complex rectangles can be built. Since rectangles have the ability to represent other regions of interest, they can also serve as predicates for querying a given dataset of regions. Taken together, this enables the creation of complex queries from rectangle operations.

Miller's algebra uses the familiar operators for intersection, union, and set difference:

$$\begin{aligned} A \cap B &\equiv \{r | r \in A \wedge r \in B\} \\ A \cup B &\equiv \{r | r \in A \vee r \in B\} \\ A - B &\equiv \{r | r \in A \wedge r \notin B\} \end{aligned}$$

For instance, if we wanted to build a new predicate *just-before* that describes a region that comes before the reference region r and ends exactly where r begins, we can define it as follows:

$$\textit{just-before}(r) \equiv \textit{before}(r) \cap \textit{overlaps-start}(r)$$

The intersection of the rectangles of *before* and *overlaps-start* results in a new rectangle *just-before*, in this case a horizontal line in two-dimensional space (cf. figure 3.8). The fact that *just-before* is again a rectangle means that it can be handled like any other region or rectangle. Consequently, it can serve as a new basis for building further predicates from it. Figures 3.7 and 3.8 list new predicates for adjacency operations that can be built in this way; figure 3.7 does so in one-dimensional space, figure 3.8 in Miller space.

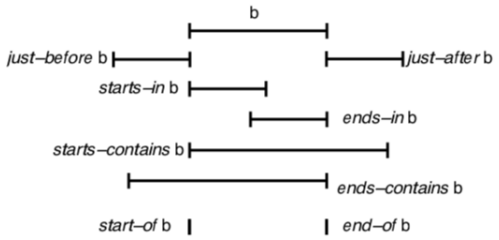


Figure 3.7: Adjacency operations (taken from Miller 2002)

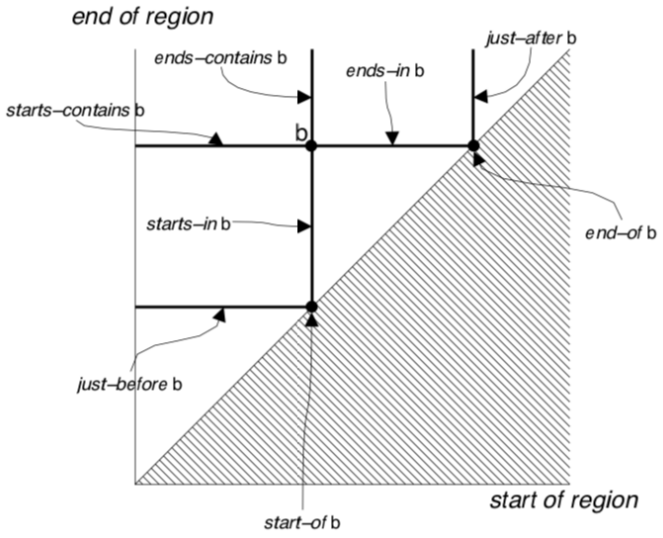


Figure 3.8: Adjacency operations (taken from Miller 2002)

Finally, let us look at what happens if we want to delete a rectangle from a set of rectangles using the set minus operator. This operation is special in that it can lead to different cases, depending on whether the respective region is fully enclosed in the other region, or whether they overlap in some way. Figure 3.9 shows this case. Depending on the specific case, this operation can lead to up to 4 new rectangles being created (as in case (a), where S fully encloses R).

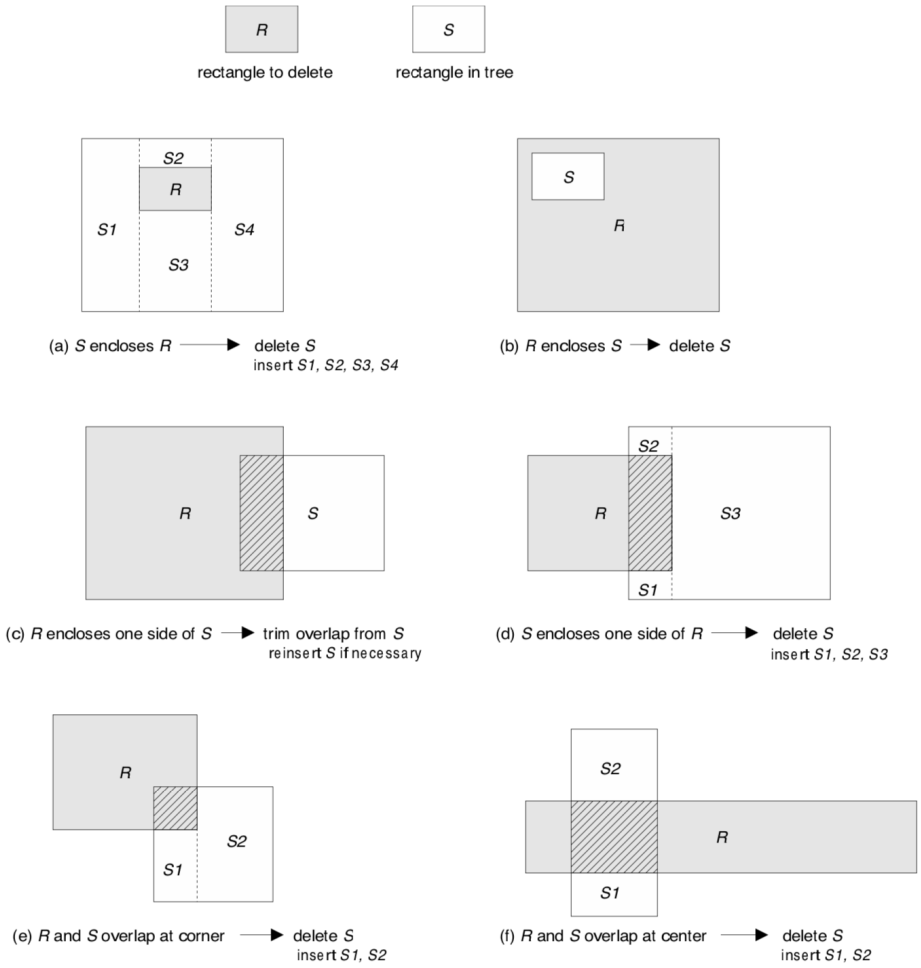


Figure 4.9: Deleting a rectangle from an RB-tree.

Figure 3.9: Rectangle deletion from index (from Miller 2002)

3.5.2 Topological notation under Region Algebra

In *codex*, I implemented the region algebra approach of regions, rectangles, rectangle set operators, and predicates, along with the query language suggested by Miller.

I will describe now how the sections expressed in my topological notation can be exactly represented as regions in Miller space, and how this can be used for the querying of topological matrices in *codex*. Remember that sections are separated from each other via synchronisation points; these define natural start and end points of sections in my sense.

Example: Querying and extracting sub-models

Given a topological notation and using *codex'* spatial query mechanisms, sub-models can be extracted from the source.

Listing 3.30 Spatial query example 1: before

```
> cat fox.codex
.          .           .                      .      .   .
           white ferret                                  | v1: b0
                                               little     | i1: b0
The quick brown fox    jumps over the lazy        dog. | b0
```

Listing 3.31 demonstrates how spatial queries can be executed by `codex-query-regions`, given the topological notation in 3.30. The query extracts every section that comes before the sections of layer `i1` and saves this predicate under the name `before-i1`.[5] This operation leads to a new matrix, containing all sections before layer `i1`'s sections, i.e. everything up to *little*.

Listing 3.31 Spatial query example 2

```
> codex query-regions --query 'before-i1 is before @i1' fox.codex
.          .           .                     .
           white ferret                      | v1: b0
The quick brown fox    jumps over the lazy   | b0
```

Listing 3.32 demonstrates the *after*-predicate. In this case, positional information is used to extract everything that comes after a certain position.

Listing 3.32 Spatial query example 3

```
> codex query-regions --query 'after-i1 is after {41..48}' fox.codex
                                                    .   .
                                                   dog. | b0
```

Example: Dealing with overlaps

Listing 3.33 demonstrates the querying for overlapping annotation. The result includes all sections that overlap with layer `a2`'s annotation region: obviously this will be layer `a1`, as well as sections from the layers `v1` and `b0`.

[5] Note that the notation `@i1` is a *codex* addition to the query language to query for a specific layer's sections.

Listing 3.33 Spatial query for overlapping structures

```
> cat fox.codex
.         .     .  .  .        .           .      .   .
          white ferret                                 | v1: b0
                                           little      | i1: b0
The quick brown fox    jumps over the lazy        dog. | b0
          --------- a1                                 | anno-1: b0
                ---------------- a2                    | anno-2: b0

> codex query-regions --query 'result2 is overlaps @anno-2' fox.codex
.         .     .  .
          white fer | v1: b0
                    | i1: b0
The quick brown fox | b0
      a1  --------- | anno-1: b0
             a2 --- | anno-2: b0
```

Miller's algebra gave us the last piece that makes *codex*' handling of concurrent hierarchies possible. The next section will describe how document model, representations, and data structures come together to allow us to perform the tasks that can be done with this document model.

3.6 The bigger picture: *codex* in the Digital Humanities world

This section explains how *codex* could be used in a real-world philology context. Initially, data enters *codex* in some form, either through an import from existing data, or by data creation from scratch, using the commandline API. At a later time, data could also be entered using a visual editor. Once data is available in the topological format, the philologist can manipulate it, annotate, do search, and interface it with other data sources from the outside. Lastly, the data can be printed out in the topological format; it can be stored away or and alos inspected in detail in this form.

As far as search is concerned, different mechanisms are supported: the philologist can search syntactically through the non-linear text, because all linearisations have already been created in the background and have been indexed for fast textual search. Secondly, search can also happen structurally, aided by the region algebra as discussed in the previous section.

Interfacing with other sources and projects can happen through export to any interchange format desired, e.g. XML or RDF. Export to XML makes XML tooling available. Note however, that such a conversion would by its very nature be lossy if there are any of the inconsistencies discussed at length in this thesis, such as overlapping annotation or non-contiguous text structures. As was discussed in chapter 2.2, it would then become

necessary to employ workarounds, so that the data can be forced back into the straight jacket of the single hierarchy model. We already know that workarounds are detrimental, as they introduce ambiguity and cannot be programmatically converted back without manual intervention. Another option is to export the data to RDF and use Semantic Web ontologies and inferencing mechanisms to enrich the data in various forms.

3.6.1 Architecture of codex

The architecture of codex is shown in figure 3.10. *Codex*' document model is the topological format. It is the logical model of how the data is prepared, formatted and internally structured so that all operations necessary can be performed with it. The logical model is so close to the visual topological format that it is basically both at the same time: the topological model is effectively a pretty-print of the logical model. If the next computing medium will be invented in the future, the API can be used to port all the data in *codex* over to the new medium. In the case of suspected long-term disruption of civilisation, the topological format could be printed. People would then actually be able to continue studying the Scholarly Editions even in a world without computers.

Internally, codex relies on a central data structure called the *section store* to realise this logical model. The section store contains, for each record, the layers that constitute it. The section store is easily convertible to other formats to support various operations, namely:

- It is convertible back and forth into the **topological format**, so that a philologist can look at the record directly or print it out. This can be combined with functionality to "wrap" long records for easy viewing (by splitting and joining records).
- If we need a linearisation of the text, the section store can be compiled into a **variant text graph (TOM and DOM)** that supports this operation. The linearisations can then be indexed for textual search.
- The variant text graph could also be used for consistency checks and to support the semi-automatic semantic validation of a record.
- If we want to query spatially, a **spatial region index** can be created. To access the store or formulate queries, the **store selector language** is given by the region algebra. The store selector language is the counterpart to XML's XPath.
- If we want to interface with the semantic web, a set of Linked Open Data (LOD) triples in RDF format can be created as a **semantic database**.

Some other input/output operations are not directly shown in the figure, namely:

- Data import from legacy formats such as XML is supported by an **API**.
- I will also show an example of **data conversion** from the Wiener Ausgabe in section 4.4.2.

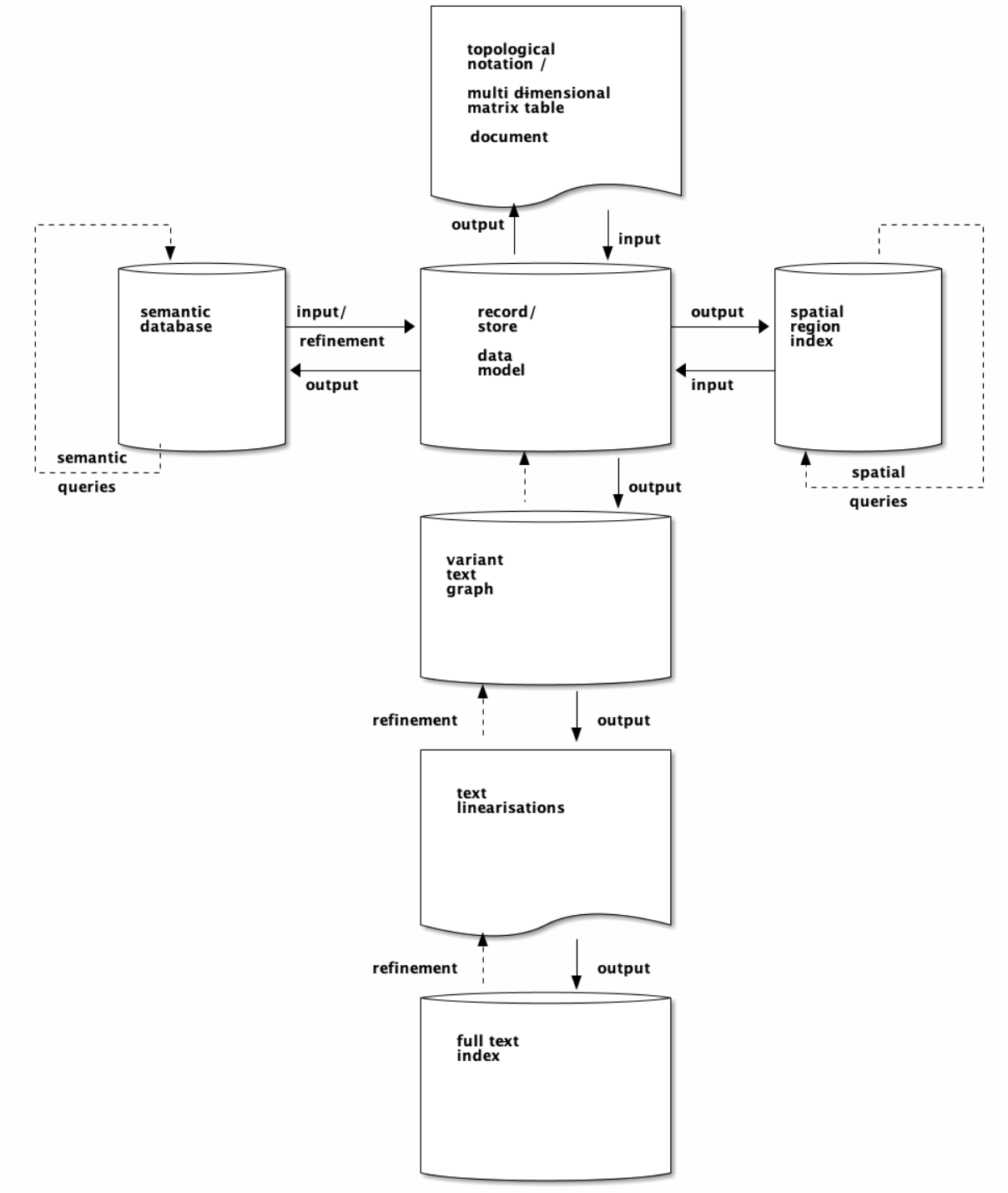


Figure 3.10: *codex* architecture

- A **command line tool** is realised directly on the API. I used it all over in this thesis for demonstrations of the functionality of *codex*. Alternatively, one could build an editor for data input using this API, for instance a graphical editor.
- A **store mutation language** allows us to manipulate data in the section store. It is the the equivalent to XSLT in the XML world. The store mutation language is a compositional language where small commands can be combined into more complex commands, comparable to pipelines in the Unix shell. Thus, it makes user-defined store mutation commands possible, also for non-experts. Internally, editorial operations and annotations are all implemented using the store mutation language.
- If we want to transform the store with external means, it can also be written out and read back in using the **JSON format**.

3.6.2 Import

In the most cases, data will not be created from scratch (at least not in the first few years of *codex*' use), but imported from existing sources. Also, some projects might want to export data from *codex* into XML regularly, for compatibility with other projects and existing tooling.

codex' document model with its spatial organisation of concurrent, synchronised layers is a successor to both the linear and the hierarchical document model, enhancing and extending them. From a practical viewpoint, this means that backwards-compatibility is possible; both types of document models are fully supported by the topological format presented here.

In this sense, import can either be flat or deep:

a) *Flat import* is a trivial operation: the foreign data format is imported as linear plain text, including all in-line markup written out character for character, into the baseline of the topological format. With flat import, it is possible to import documents expressed in either the tree or the linear format into topological notation without compromising the information contained in original document. In the flat import scenario, the XML source is not interpreted in any way and treated like plain text. A flat import means that we take the original source as it is straight into the baseline and leave it untouched. This means that any legacy format or document can be annotated without comprising the original source.[6]

[6]Remarkably, flat imports can still benefit from the predicates defined by Miller (Miller 2002): Miller provides predicates that treat the contained tags hierarchically (*descendant-of*, *ancestor-of*) and match opening and closing pairs of delimiters (*balance*). There are also predicates that ignore meta-data (i.e. the XML tags) altogether and treat it as whitespace.

We are free to add new annotation, and this will be represented as separate layers. The original source is not touched in this operation. On flat import however, we lose the ability of meaningful long-term storage: the legacy documents (which are now only represented on the baseline) remain as they were, potentially convoluted and unreadable to the human.

b) Foreign formats can also be structurally converted through a *deep import*, which is an additional operation after a flat import. Using a deep import allows us to import a format into *codex* proper; here, we gain a real topological notation and all its advantages, including meaningful long-term archiving. In the case of XML this means that the linear text on the baseline is stripped of markup, and turns into a clean new baseline of only text. The stripped metadata is then programmatically imported into annotation layers of the topological format. Additionally, if appropriate schemas are given (such as XML/TEI), particular meta-data of the original source can even be lifted from annotation layers into editorial operation layers on top of the baseline. To a certain degree, it might even be possible to repair idiosyncratic tree-structured documents automatically upon import, although in the general case, the import of idiosyncratic documents necessarily requires human intervention. Deep import is not fully implemented yet and currently exists in a proof-of-concept stage.

Note that these mechanisms should in principle work for any machine-readable format in any markup language, not only for XML.

I have just discussed deep import from XML, which should be sufficient for many cases. However, sometimes we want to input data from more complicated formats. An example of such a more complicated format is the Wiener Ausgabe, which has aspects that go against the deep import paradigm. In order to input data from typesetting formats, we need to perform a *conversion*, a more involved operation. I will give an example in section 4.4.2.

3.6.3 Export

Besides rendering fully-linearised sources for indexing search engines, *codex* allows export to different formats. Using *codex*' export facilities makes the data available in other formats and thus provides access to existing tooling and workflows, such as typesetting or data transformations. Together with the import facilities, *codex* can serve as the central data repository from which one can go in different directions.

There are two main reasons why export can be useful:

- **Export makes it possible to use existing tooling**. For instance, if we export to XML, we have access to XML tooling such as the established XSL transformation technology. An established project might switch to *codex*, but still keep legacy XML

pipelines, e.g. for rendering parts of their website by using their established XSL transformation pipelines. This of course comes with the caveat that if overlapping structures are present in the data, then one of the workaround strategies described in Section 2.2 (milestone, standoff, or artificial fragmentation) must be chosen; this will be a lossy and thus harmful operation and one cannot guarantee automatic reversion of the workaround.

- **Export makes new tooling available**. For instance, if we export to RDF, we have access to an extended form of a database, including inference. In the words of Berners-Lee et al. (2006), "giving users the ability to link pieces of texts to their representation elsewhere (e.g. on the Web) opens interesting possibilities for data integration, automatic reasoning and semantic analysis in the Linked Open Data". In the framework of my DH project, the vision would be to support queries such as the following: "Show me all remarks where Wittgenstein crossed out 'philosophy' using red ink and substituted it for something else. Furthermore, show me all the manuscripts where this happens and the archives which keep them."

codex allows export to different formats: XML, RDF, and JSON.

The export to JSON, at the simplest level, can serve as a central piece of data exchange with different applications. It is thus part of the API which other applications can use to interact with the data model. A visual editor would exchange a JSON export of *codex*' data and vice versa, *codex* can receive JSON and build a topological representation from it (cf. *codex*' architecture on page 89).

To realise the vision of an extended form of a database mentioned above, we need to export to RDF, link it with an ontology, and then use a reasoner to draw inferences on the data.

A good framework for an RDF export would be Extreme Annotational RDF Markup (Peroni, Vitali, and Di Iorio 2009; Peroni and Vitali 2009; Peroni, Gangemi, and Vitali 2011; Di Iorio, Peroni, and Vitali 2010; Barabucci, Peroni, et al. 2012, EARMARK;), "a unifying framework that can handle tree-based XML features and also more complex markup for non-XML scenarios such as overlapping elements, repeated and non-contiguous ranges and structured attributes". Once my data is in RDF, Linked Open Data (LOD) tooling[7] would be available. Linked Open Data offers an extended form of a database which can help users explore and analyse the data, and thus allows us to go from from documents to data. Semantic web ontologies can then be used together with automated reasoners.

[7] "Linked Open Data" is new terminology for what used to be known as the Semantic Web. Although research in Linked Open Data is not as active as it used to be, it still is used widely in some research areas (e.g. biology). For a good, quick introduction to Semantic Web Technologies the reader is referred to Horrocks 2008.

One would then choose or build a suitable ontology that serves to model the actual needs of the users. Ontologies can act as a middle ground between philologists, philosophers, linguists and computer scientists, helping both computer experts and non-computer experts in building, maintaining, developing and integrating heterogeneous data sources (Wang et al. 2009). In case we have more than one project and more than one ontology, one can even create mappings between them.

Inferencing allows us to arrive at novel pieces of knowledge that were not explicitly put into the database, but which can be deduced from the existing data; an example for such inference are the Aristotelian syllogisms. For querying, one could then use SPARQL (Seaborne and Prud'hommeaux 2008; Ferré 2010). Visualisation is another area where data could be enriched once it is in RDF format. Non-expert users in particular can profit from methods for information perception and manipulation. One example for visualisation is SynopsViz, a tool for hierarchical charting and visual exploration of Linked Open Data (Bikakis, Skourla, and Papastefanatos).

This concludes my description of the design of codex. In the following chapter, I will describe implementational details of this work.

Chapter 4

Implementation

This chapter documents the implementation of *codex*; the data structures and algorithms that realise the data model, as well as the syntax and validation mechanisms designed in the previous chapter.

4.1 Section Store

The section store consists of instances of the topological format, also called blocks. One block of the topological format is called a *record*. A record consists of *layers* (corresponding to the rows of the topological notation). Layers consist of *sections* (corresponding to areas between synchronisation points). The section store itself contains a sorted set of all sections of a record.

4.1.1 Data Structure

The central piece of information is the *section*. A *section* represents a span of text inside a layer. The section data structure is defined as a tuple of the following objects (cf. listing 4.1):

- a *type* indicating the section's type (e.g. insertion or annotation)
- the text *data* contained in the section; which can be empty, e.g. in the case of deletion
- the start *position* of the section
- the *length* of the section; the length can be longer than the actual text data contained in it, e.g. in the case of a variant
- a layer *id*, indicating the layer the section belongs to
- a parent layer *pid* of the current layer, for all layers except the baseline
- an *author* (or a placeholder field for additional metadata)

Listing 4.1 Section data structure

```
struct Section {
    SectionType type;
    string_type data;
    std::size_t pos;
    std::size_t len;
    string_type id;
    string_type pid;
    string_type author;
};
```

A *section store* (listing 4.5) keeps a list of all sections and establishes a total order over the sections. The total order between sections is defined by a scaffolded comparison of its values (listing 4.2), while the equivalence relation is defined by the equivalence of the individual tuple elements (listing 4.3).

Listing 4.2 Order relation between Sections

```
bool operator<(const Section& a, const Section& b) {
    return std::tie(a.type, a.pos, a.len, a.author) <
           std::tie(b.type, b.pos, b.len, b.author);
}
```

Listing 4.3 Section equivalence relation

```
bool operator==(const Section& a, const Section& b) {
    return std::tie(a.type, a.id, a.pid, a.pos, a.len, a.author) ==
           std::tie(b.type, b.id, b.pid, b.pos, b.len, b.author);
}
```

Canonicalisation is an operation where the store's contents are written out as a topological notation. Remember that annotation layers are situated below the baseline, while editorial operations are placed above it. Therefore, we need to know a section's type during canonicalisation. There are 9 section types, as given in listing 4.4.

Listing 4.4 Section Types

```
enum SectionType {
  NONE, PAUSE, TEXT, SUB, DEL, VAR, INS, BASE, ANN
  // 0    1      2    3    4    5    6     7    8
```

If what we are dealing with is a textual layer expressing a substitution (type `SUB`), the insertion part is made to sit directly above the deletion and is correctly aligned. Note that the order in which the sections should be stacked in the topological notation are predetermined by their order in the *enum* datastructure, to make such operations easy.

Per se, layers do not exist as an entity of their own. Instead, a layer is formed by accumulating all sections with the same `id` from the section store. This operation is called a projection. Line 6 of listing 4.5 shows the constant method `layers()` which is used to project layers from the section store.

4.1.2 Section Store Implementation

The section store is implemented as a vector of sections. It builds the cornerstone around which everything else is built.

Listing 4.5 Section store data structure

```
1  struct Store {
2    using value_type = std::vector<Section>;
3    using fifo_map   = nlohmann::fifo_map<std::string, Layer>;
4
5    std::size_t           width() const;
6    fifo_map              layers() const;
7    std::set<std::size_t> syncros() const;
8
9    void apply(Command*);
10   void sort();
11
12   void join(const std::vector<Store>&);
13
14   Store query(const string_type& query);
15
16   friend std::ostream& operator<<(std::ostream&, Store);
17   string_type to_string();
18
19   value_type sections;
20 };
```

Centrally, the section store keeps an list of `Section` objects (lines 2 and 18). It also provides a sorted map of layer identifiers and layers (line 6). This list of layers is built dynamically from the sections' stored *id* information. Similarly, the resulting matrix' width as well as the synchronisation points are calculated upon request (lines 6 and 7). Note that in this implementation the section store does not keep a list of sorted sections at all times, but instead sorts the list of sections when needed (line 10).

Furthermore, the section store can be modified by applying `Command` objects (from the Section Store Mutation Language, line 9), and also be joined with other section stores (line 12), as well as be printed to a topological matrix (line 16). The section store can also be queried using the Section Store Query Language (line 14), which is based on Miller's rectangles and set operations, described in section 3.5.2. The implementation behind it will be described in section 4.3.1.

Practical Example: Section Store's contents

The ordered set of sections, together with the Section types enables the automatic stacking and correct alignment of the sections as a matrix, which is the output. Listing 4.6 shows a simple section store.

This set of sections results in the topological notation of listing 4.7.

Listing 4.6 Store example as a JSON set of Section objects

```
[
  { pos:  4, len: 6, data: "quick ",    id: "i1", pid: "b0", author: "", type: INS },
  { pos:  0, len: 4, data: "The ",      id: "b0", pid: "",   author: "", type: BASE },
  { pos: 10, len: 9, data: "brown fox", id: "b0", pid: "",   author: "", type: BASE }
]
```

Listing 4.7 Equivalent Topological Notation

```
.   .     .        .|
    quick           | i1: b0
The       brown fox | b0
```

4.1.3 Serialisation and deserialisation of records

The topological notation format and the document model behind it are very closely connected. Since there is a total order on the sections in document model, it is easy to build the corresponding topological notation from it, via layer projection.

Listing 4.8 Store print algorithm

```
1  string_type Store::to_string() {
2      string_type str;
3      sort();
4
5      auto layers  = fifo_layers();
6      auto syncros = this->syncros();
7      auto width   = this->width();
8      auto ruler   = Ruler(width, syncros);
9
10     str += ruler.to_string();
11
12     for (const auto& [id, layer] : layers) {
13         str += layer.to_string(width, syncros) + "\n";
14     }
15
16     return str;
17 }
```

Listing 4.8 outlines the algorithm for the serialisation of a section store into a topological matrix layout:

- Initially, the store is sorted (this is a trade-off, so the store is only ever sorted upon serialisation).
- Next, layer projections are fetched into a `fifo_map`, together with a list of the synchronisation points of the store's sections, as well as the matrix' output width (which can depend on the `data` as well as `len` atttributes of the sections).
- The ruler (the top bar indicating the synchronisation points) is printed.
- The layers are printed one by one to the target-width of the matrix (listing 4.9 shows the layer's print algorithm).

Next we consider the print algorithm for a layer, which is given in listing 4.9. This

algorithm takes care of the necessary paddings. Padding becomes necessary when the target-length of a section is bigger than the actual data content.

Listing 4.9 Layer print algorithm

```
string_type Layer::to_string(std::size_t width, std::set<std::size_t> syncros) const {
  const char LPAD = ' ';
  const char RPAD = ' ';

  string_type line;
  if (this->is(SectionType::ANN)) {
    for (auto i = 0u; i < sections.size(); i++) {
      auto& section = sections.at(i);

      line += string_type(section.pos - line.utf8().length() -
                                section.data.utf8().length() - 1, LPAD);
      line += section.to_string();
    }
  } else {
    line = std::accumulate(  //
        std::begin(sections), std::end(sections),
        string_type(sections.front().pos, ' '),
        [](string_type acc, Section cur) {
          if (cur.pos > acc.utf8().length())
            acc += string_type(cur.pos - acc.utf8().length(), ' ');
          acc += cur.to_string();
          return acc;
        });
  }

  if (width > line.utf8().length())
    line += string_type(width - line.utf8().length(), RPAD);

  line += metadata();

  return line;
}
```

Deserialisation of records corresponds to parsing the topological notation records. This is handled through a dedicated reader class, `codex::reader::CodexBlock`. Listing 4.10 shows the individual parsing steps.

Listing 4.10 Parsing a codex block

```
bool CodexBlock::parse() {
    parse_syncros();
    match_layers();
    tag_baseline_sections();
    tag_annotation_sections();
    tag_text_operations();
    tag_complex_text_operations();
    add_sections();
    Store store{sections};
    return true;
}
```

Essentially, the parsing of the topological notation of the codex block is a multi-pass algorithm with the following steps:

- The ruler headline is parsed to determine the synchronisation points.
- The layers are read into a vector and metadata is parsed. In this step, `id`s and `pid`s are collected. This step allows us to build the DOM.
- Next, the baseline is identified: the baseline is the central piece of information, telling us which layers are textual layers and which are annotation layers. The baseline is easy to find, since it is the only layer without a parent id.
- In a first pass, all textual and annotational layers are tagged as such. In this step, the following three essential section types are identified:
  1. empty sections which do not carry textual content and which are only there for padding purposes;
  2. sections which represent deletions (these are easily identified by a series of -symbols);
  3. sections carrying text.
- But tagging the sections in this way is not enough. For now, we have only identified whether a section contains text or is empty in its data. If a section does contain text, up to this point, we can not distinguish an insertion from a variant. In order to resolve this, we need information from the DOM about the parentship of layers: for instance, a non-empty section that is aligned with an empty section on the parent layer can now be interpreted as an insertion. Only in this second pass with additional information from the DOM can sections be correctly interpreted in their topological arrangement. This follows from the core principle that symbols in the topological notation gain their meaning from their topological arrangement.
- Finally, complex textual operations such as a substitution are evaluated. This step searches the DOM graph for combinations of deletion/insertions patterns that correspond to specific complex textual operations across several layers.
- In a last step, the parsed sections are added to the section store.

4.1.4 Reformatting records

Records can be reformatted in different ways: they can be concatenated, split, re-joined after a split and wrapped at user-specified column widths. Concatenating records is the task of joining two topological notation matrices. Given my definition of a section (listing 4.11, repeated here for convenience), this task becomes trivially executable.

In order to join two records, a new section store is built, which contains all sections of both stores. Then, the second store's begin positions (`pos`) are incremented by the width of the first record.

Listing 4.11 Section data structure

```
struct Section {
    std::size_t pos;
    std::size_t len;
    string_type data;
    string_type id;
    string_type pid;
    string_type author;
    SectionType type;
};
```

Splitting records can be useful in contexts where a record is to be fit to a certain width due to space constraints of the output medium (could be a screen or paper). This is similar to the line-wrapping performed on monitors or by printers (for instance, to 80 columns). Line-wrapping by itself is not the entire solution here, since the matrix document model consists of several synchronised lines. More attention is therefore necessary.

Using the store mutation language, sections are split using `split_sections_at`, an operation that is followed by a correction of the begin positions (`pos`) of the store's sections. So all in all, splitting a record is almost as trivial as joining records.

Practical Example

I will now demonstrate split and rejoin operations. Consider Listing 4.12, which is a copy of Listing 3.1 we have already encountered on page 56. It is so wide it barely fits on the page.

Listing 4.12 Topological notation

```
.   .             .          .   .   .       .    .        .          .      .    .      .|
                                  let us fathom     vast                                   | r3: b0
Such               ----------                                                            | r2: r1
----               like these                                                            | r1: b0
Such considerations           can show us       the infinite  possibilities of the means of our language | b0
```

Listing 4.13 demonstrates how a large block such as this can be split at a user-specified column position.

The only parameters we need for a split is the desired width (`--width`), here 38 characters. We receive two records printed in one file, with blank lines separating them. In order to economise on the cognitive load of the users, the second block now only contains layer `var1` and the baseline; layers `del1` and `ins1` are omitted because they contribute no relevant information for the split block. *Codex* has intentionally removed these to allow for a maximally compact representation of each block. Likewise, layer `var1` has been removed from the first block.

Listing 4.13 Basic API functionality: splitting blocks

```
> codex split --width 38 such-considerations.codex
.   .              .           .   . .
----                                 | del1: b0
                    like these       | ins1: b0
Such considerations             can  | b0


.             .   .       .              .
let us fathom     vast              [...] | var1: b0
show us       the infinite multitude [...] | b0
```

I also provide code for re-joining of smaller blocks which are the output of the previous split. This is shown in Listing 4.14.

Listing 4.14 Basic API functionality: joining blocks

```
> codex format --join such-considerations.codex
.   .              .           .   .               .   .        .               .
----                                                                [...] | del1: b0
                    like these                                      [...] | ins1: b0
                                       let us fathom     vast           [...] | var1: b0
Such considerations             can show us       the infinite multitude [...] | b0
```

If given an input file such as the one in listing 4.13, which resulted from a split, *codex* can revert the split and thus create a single record from these[1].

It does so by collecting all layers from all blocks, synchronising them and printing a version with the union of all layers.

For instance, in listing 4.13 the first block of the split blocks contains layers `del1` and `ins1`, while the second block does not. The resulting big block will need to contain all three of these layers, as well as the baseline.

4.1.5 Store mutation

Users need pragmatic ways to interact with and transform the section store. Users want to perform operations such as adding insertions, annotating, splitting and joining blocks. They may also want to add new store mutations. Such store mutation happens through the store mutation commands, which are equivalent to XSLT in the XML world. Store mutation can be used to transform the data model and to derive a new model. Users are free to add their own mutation commands using the store mutation commands.

Editorial operations are internally also implemented as mutation commands. Mutation commands traverse the store's sections, transform these and return a new store. For

[1]Note that this operation requires as input a file that is the exact output of a codex split. This ensures that the file does not contain any clashes in sections' identifiers or associated contents. The join will not work with arbitrary, user-patched files that vaguely look like the one in listing 4.13.

instance, let's assume an initial section store like the one in listing 4.15 with only one section.

Listing 4.15 Store mutation example

```
[
  {
    pos: 0, len: 13, data: "The brown fox", id: "b0", pid: "", type: BASE
  }
]
```

If a user executes the command: `codex add-insertion --data "quick " --at 4` on the commandline, then the following steps will happen in the background. One can model an insertion by using the following basic mutation commands, in sequence:

- `split_sections_at` splits all sections which happen to cover the insertion position into two separate sections. This also creates a new synchronisation point. This is exemplified in listing 4.16.

Listing 4.16 Store mutation example step 1

```
[
  { pos: 0, len: 4, data: "The ",      id: "b0", pid: "", type: BASE },
  { pos: 0, len: 9, data: "brown fox", id: "b0", pid: "", type: BASE }
]
```

- `move_sections_after` moves all sections which occur after the newly created split position from the previous step, and does so by exactly the amount necessary for the insertion to fit. This is exemplified in listing 4.17.

Listing 4.17 Store mutation example step 2

```
[
  { pos:  0, len: 4, data: "The ",      id: "b0", pid: "", type: BASE },
  { pos: 10, len: 9, data: "brown fox", id: "b0", pid: "", type: BASE }
]
```

  Note how the newly created split section was moved right. The second part now starts at position 10. Space was created, namely exactly the length of the newly created text '`quick `', including the blank, i.e. 6 more characters, plus 4 for the insertion's position.

- `insert_section` adds the insertion section itself to the section store. This is exemplified in listing 4.18.

  This now results in the desired notation, shown in listing 4.19.

Store mutation commands can be combined using the pipe symbol (in line with Unix tradition), in order to form more complex, self-defined commands, where

Listing 4.18 Store mutation example step 3

```
[
  { pos:  4, len: 6, data: "quick ",    id: "i1", pid: "b0", type: INS },
  { pos:  0, len: 4, data: "The ",      id: "b0", pid: "",   type: BASE },
  { pos: 10, len: 9, data: "brown fox", id: "b0", pid: "",   type: BASE }
]
```

Listing 4.19 Equivalent Topological Notation

```
.   .     .        .|
    quick           | i1: b0
The       brown fox | b0
```

each resulting output is passed on as input to the next command in succession: `split_sections_at | move_sections_after | insert_section`.

4.2 Variant Graph

Codex supports indexed full-text search. In order to accomplish this, all possible variants of the multi-linear text need to be linearised. The role of the DOM and TOM structures for linearisation has been described in section 'Practical Linearisation Example' on page 63. In the following, the index structures behind full-text indexing are explained. The graph implementation itself builds on the Boost Graph Library.

Both the TOM and the DOM build on a common graph structure, which is shown in listing 4.20.

The common graph data structure is templated with regards to the type of vertex[2] as well as edges types, as can be seen in line 1. This allows it to serve for different graph implementations which the TOM and DOM will build on. It offers various methods to ease the respective implementations, such as methods to add vertices and edges (lines 24 through 27), to retrieve all vertices and edges (lines 18 and 19), and to determine the number of vertices and edges (lines 15 and 16). Lastly, it can be exported to the graphviz graph language to visualise the structure, as shown in listing 4.21. This approach has been used to produce all the graphs in this thesis.

[2]The code uses the term "vertex" instead of "node" in accordance with the naming in the Boost Graph Library. I will use the term "vertex" in the following.

Listing 4.20 DAG implementation

```
template <typename VertexType, typename EdgeType>
struct Graph {
  using vertex_type       = VertexType;
  using edge_type         = EdgeType;
  using graph_type        = boost::adjacency_list<  //
      boost::vecS,
      boost::vecS,
      boost::bidirectionalS,
      vertex_type,
      edge_type>;
  using vertex_descriptor = typename graph_type::vertex_descriptor;
  using edge_descriptor   = typename graph_type::edge_descriptor;
  using vertex_list       = std::vector<vertex_descriptor>;

  auto num_vertices() const noexcept;
  auto num_edges() const noexcept;

  auto vertices() const noexcept;
  auto edges() const noexcept;

  std::set<vertex_descriptor> start_states() const noexcept;
  std::set<vertex_descriptor> final_states() const noexcept;

  auto add_vertex(std::optional<vertex_type> v = {}) noexcept;

  void add_edge(vertex_descriptor u, vertex_descriptor v) noexcept;
  auto add_edge(vertex_descriptor u, vertex_descriptor v, edge_type e) noexcept;

  void write_graphviz(std::ostream& = std::cout) const noexcept;

  graph_type G;
};
```

Listing 4.21 GraphViz dot

```
digraph {
  rankdir = LR

  0 -> 24 [label="Such considerations can"]

  24 -> 38 [label="show us"]
  24 -> 38 [label="let us fathom"]

  38 -> 42 [label="the"]

  42 -> 51 [label="infinite"]
  42 -> 51 [label="vast"]

  51 -> 98 [label="multitude of the possibilities of our language"]
}
```

4.2.1 The Document Object Model (DOM)

The Document Object Model (DOM) tracks the interdependencies between layers and thus all possible compositions of layers. We have seen an example DOM in figure 3.3 on page 65. The definition of the `DOM` structure is presented in listing 4.22.

Listing 4.22 DOM implementation

```
1  struct DOM {
2    struct Edge {};
3
4    using layer_type        = std::shared_ptr<Layer>;
5    using graph_type        = Graph<layer_type, Edge>;
6    using vertex_type       = graph_type::vertex_type;
7    using vertex_descriptor = graph_type::vertex_descriptor;
8    using id_type           = Layer::id_type;
9    using map_type          = std::map<id_type, vertex_descriptor>;
10
11   DOM(const Store&);
12
13   void insert(const layer_type&) noexcept;
14   void connect_layers() noexcept;
15
16   std::optional<vertex_descriptor> find_by_id(const id_type&) const noexcept;
17
18   void write_graphviz(std::ostream& = std::cout) const noexcept;
19
20   auto&       graph() noexcept { return graph_; }
21   auto const& graph() const noexcept { return graph_; }
22
23   graph_type graph_;
24   map_type   map_;
25 };
```

The DOM's vertices represent layers, whereas the edges, which are directed, are empty objects.

The DOM builds on the previously shown basic graph structure to model the interdependencies between layers (lines 4 and 5). A DOM can be constructed from a section store (line 11) using the methods `insert` and `connect_layers` to build the DOM structure.

The crucial algorithm part for the DOM is its construction from the section store through the methods `insert` and `connect_layers`. To construct a DOM from a section store, method `insert` initially creates a node for each layer. Each node is named after the layer's `id`. Next, `connect_layers` looks at each node in turn and tries to find its parent node through the respective layer's `pid` (using method `find_by_id`). The parent node is now connected to the original node through a directed edge.

4.2.2 The Text Object Model (TOM)

Recall the example TOM given in figure 3.2 on page 64 as a visualisation of the Text Object Model graph. Like the DOM, the TOM builds on the basic graph structure in its implementation.

Listing 4.23 TOM implementation

```
1  struct TOM {
2    struct Vertex {};
3    struct Edge {
4      std::shared_ptr<Section> section;
5    };
6
7    using id_type           = Layer::id_type;
8    using pos_type          = std::size_t;
9    using section_type      = std::shared_ptr<Section>;
10   using vertex_type       = Vertex;
11   using edge_type         = Edge;
12   using graph_type        = Graph<vertex_type, edge_type>;
13   using vertex_descriptor = graph_type::vertex_descriptor;
14   using edge_descriptor   = graph_type::edge_descriptor;
15   using vertex_list       = std::vector<vertex_descriptor>;
16
17   TOM(const Store&);
18
19   void write_graphviz(std::ostream& os = std::cout) const noexcept;
20
21   graph_type                                graph;
22   std::map<std::size_t, vertex_descriptor> vertex_map;
23 };
```

Let's now look at the TOM implementation in listing 4.23. Initially, the Text Object Model (TOM) graph structure is constructed from the Section Store. This can be seen in line 17. Initially, the TOM is constructed by creating a node for each synchronisation point of the topological notation. Next, edges between the newly created nodes are created for each section of each layer. In the TOM, nodes are empty objects, whereas edges carry section information, namely the section's text data, together with the layer id of the originating layer (cf. figure 3.1 on page 64).

The TOM implementation is memory-efficient in that the edges do not carry the actual section information but rather pointers into the section store from which the TOM is built. Methods to perform linearisations are handled by a dedicated class `Linearizer` using a TOM structure, which I will next describe.

4.2.3 Linearisation

A specialised `Linearizer` class takes care of building linearisations using information from the TOM and the DOM. The algorithm for linearisation is given in listing 4.24.

Listing 4.24 Linearisation algorithm

```
std::vector<Linearization>
Linearizer::linearizations(const std::vector<id_type>& tokens) {
  std::vector<Linearization> linearizations;

  for (auto id : tokens) {
    const auto walker = [&](auto edge) {
      return tom.graph.G[edge].section->id == id;
    };

    const auto paths = tom.graph.all_edge_paths(0, walker);

    for (auto path : paths) {
      linearizations.emplace_back(Linearization{id, ""});

      for (auto edge : path) {
        linearizations.back().data += edge->data;
      }
    }
  }

  return linearizations;
}

template <typename VertexType, typename EdgeType>
std::vector<typename Graph<VertexType, EdgeType>::edge_list>
Graph<VertexType, EdgeType>::all_edge_paths(
    vertex_descriptor start, predicate_type predicate) const noexcept {

  std::vector<edge_list> paths;

  for (auto edge : out_edges(start))
    all_edge_paths(edge_list{}, edge, paths, predicate);

  return paths;
}

// recursive method
template <typename VertexType, typename EdgeType>
void Graph<VertexType, EdgeType>::all_edge_paths(
    edge_list               path,   // the current path
    edge_descriptor         edge,   // the current edge
    std::vector<edge_list>& edges,  // the path collection
    predicate_type          predicate) const noexcept {
  if (not predicate(edge))
    return;
  else
    path.emplace_back(G[edge].section);

  if (auto out = out_edges(target(edge)); out.size() == 0)
    edges.emplace_back(path);
  else
    for (edge_descriptor next : out)
      all_edge_paths(path, next, edges, predicate);
}
```

Linearisation of a reading takes place by TOM traversal. As input, it requires a specific layer, given by the layer's `id`, together with the inheritance chain of parent `id`s, which are collected from the DOM. Given this list of allowed tokens, the TOM is traversed recursively, in depth-first order, using the list of allowed tokens.

During the traversal, each out-going edge's `id` is checked whether it is contained in the token set. If so, the edge is collected in a `List<List<Edge>>`. The graph is followed through to a final state in the same manner until there are no more out-going edges. At the end of the traversal, the algorithm backtracks to the last point of diversion recursively collecting more linearisations, until there are no more backtracking-points. Finally, after concatenating all edges' text data, the `List<List<Edge>>` is flattened into a `List<String>`, representing all possible linearisations.

Example: Full-text indexing using symmetric graph structures

The following practical example shows full-text linearisation for symmetric full-text indexing using SIS (Symmetric Index Structures; Bruder 2012), which itself is based on research by Gerdjikov et al. 2013a and Gerdjikov et al. 2013b. SIS is represents a Symmetric Compacted Directed Acyclic Word Graph (SCDAWG). The symmetric nature of the index allows dynamic navigation in full-text search with immediate auto-completion in both reading directions. Initially, all linearisations are dumped into a text file. Then, this text file is indexed into a SCDAWG structure and queried, returning all linearisations containing the queried item.

```
> codex dump-linearizations such-considerations.codex > fulltext.idx
Such considerations can show us the infinite multitude... [b0]
Such considerations can let us fathom the vast multitude... [b0, var1]
Considerations like these can show us the infinite multitude... [b0]
Considerations like these can let us fathom the vast multitude... [b0, r1]

> sis query --width 40 --query "vast" fulltext.idx
      Such considerations can let us fathom the |vast| multitude... [b0, var1]
Considerations like these can let us fathom the |vast| multitude... [b0, r1]
```

4.3 Region Algebra

4.3.1 Spatial Index Implementation

The Section Store can be converted into a spatial region index for the querying, extraction and re-shaping of topological notation blocks. The spatial region index (listing 4.25) builds

on the Boost Geometry[3] and and Boost Polygon[4] libraries. The Boost Geometry library provides a R*-Tree structure implementation, which is a special data structure for the indexing and querying of spatial information. My Region Algebra implementation uses the R*-Tree index structure to index sections from the Section Store. Rectangles are built on top of Boost Polygon which provides the rectangle concept for the implementation[5]. The rectangles behave exactly as defined by Miller (Miller 2002) and as outlined in section 3.5.2.

As described in section 3.5.2, Miller (2002) defined how to use rectangles as query predicates, and how predicates can be combined to build sophisticated queries. In order to do this, we need to support the operations of rectangle union, rectangle intersection and rectangle difference. Boost Polygon implements the Manhattan and Sweeping algorithms and provides the set operations for the union, intersection and difference of rectangles.

Boost Geometry implements a powerful query mechanism on the R*-Tree index structure. Among other things, this allows to query the index for every region that is covered by a query predicate, i.e. a rectangle.

Listing 4.25 Implementation of the spatial index structure

```
struct Index {
private:
  using predicate_type = RegionSet;
  using index_type =
      boost::geometry::index::rtree<Section,
                                    boost::geometry::index::rstar<16>,
                                    indexable_getter_type>;

  index_type index_;

public:
  auto query(const predicate_type& rectangles) const {
    Store idx;
    for (const auto& rectangle : rectangles)
      std::for_each(index_.qbegin(rectangle), index_.qend(),
                    std::back_inserter(idx.sections);
    return idx;
  }
};
```

Listing 4.25 shows an excerpt of the index's implementation. Centrally, `struct Index` uses a `boost::geometry::index::rtree` data structure to index `Sections`, which provides the query functionality. The `query`-method accepts a set of rectangles and checks the index for sections that are covered by one of the predicate-rectangles. Finally, the `query`-method returns a new section store.

[3] https://www.boost.org/doc/libs/1_79_0/libs/geometry/doc/html/index.html
[4] https://www.boost.org/doc/libs/1_79_0/libs/polygon/doc/index.htm
[5] The Boost Polygon Library offers an adaption to seamlessly work together with Boost Geometry.

Using Miller's query predicates, the spatial index can thus be queried for hierarchical as well as overlapping annotation.

4.3.2 Store access via Store Selector Query Language

Store access consists of two parts, *a)* the *Region Algebra* implementation and *b)* the *Store Selector Query Language*, which is a domain-specific language. In addition to the region algebra, my implementation also includes the query language suggested in Miller's work (Miller 2002). My query language design closely follows Miller's suggestion, which was specifically crafted to make approachable data access possible for non-experts, i.e., users who are not trained in computer science. The implementation of the Store Selector Language in *codex* uses the Boost Spirit X3 library for the parsing of selection expressions. The Store Selector Query Language makes programmatic extraction and querying of sub-parts of a record possible. It represents the equivalent to XPath in the XML paradigm.

I will now show in a practical example what this accomplishes.

Listing 4.26 Building a Store and querying an Index

```
1 Baseline b0 = Baseline{"This long base line.", {"b0"}};
2 Substitution s1 = Substitution{"LONG", 5, 4, "s1", {"b0"}};
3
4 Store store = Store() | add_baseline(b0) | add_substitution(s1);
5
6 Index idx{store};
7
8 auto result = idx.query(overlaps(Region{0,10}))
9 std::cout << result << std::endl;
```

Listing 4.26 shows how a store can be built by combining commands from the Store Mutation Language (line 4), how to build an index (line 6), and how this store is then queried for overlapping regions. In this example, a baseline is added, and a substitution is performed. The section store is then indexed. Using the C++ API, it can now be queried for regions overlapping any query region from `{0..10}`.

Of course these internal routines are matched by a commandline API on the user level; we have already encountered this in section 3.5.2 on page 86.

4.4 Import and Data conversion

In section 3.6.2 I explained the design for import and export into and from *codex*. In this section, I will describe the state of implementation of this.

4.4.1 Deep Import

Remember that deep import is the operation where a document is imported into *codex* proper. This means that the original document is emptied of its markup: depending on the markup's nature, annotation is integrated into the annotation tier or, given a document schema, editorial operations of the original format are turned into editorial operations in *codex*.

Consider listing 4.27, which shows an excerpt from a TEI-annotated source of our long-running example.

Listing 4.27 Original XML to be deep-imported

```
1 <ab><seg><s>
2 ...can show us the 
3 <choice type="dsl">
4 <orig type="alt1">infinite</orig>
5 <orig type="alt2">vast</orig>
6 </choice> multitude of the means of our <lb/> language...
7 </s></seg></ab>
8 ....
```

For instance, a variant (namely between *infinite* and *vast*) has been expressed by the use of a `choice` tag (on line 3 of listing 4.27): this is the TEI-way to express an open variant.

Listing 4.28 Deep import of XML into codex (full display)

```
.                 .       .                                          .
                  vast                                                 | v1: b0
...can show us the infinite multitude of the means of our language...  | b0
---------------------------------------------------------------------- ab  | l1: b0, v1 [xml]
---------------------------------------------------------------------- seg | l2: l1     [xml]
---------------------------------------------------------------------- s   | l3: l2     [xml]
                  -------- choice type="dsl"                           | l4: l3     [xml]
                  -------- orig type="alt1"                            | l5: l4     [xml]
                  ---- orig type="alt2"                                | l6: l4, v1 [xml]
                                                 ^ lb                  | l7: b0     [xml]
```

Listing 4.28 shows the result of a deep import of the material in listing 4.27. Firstly, all editorial operations have been resolved to their proper topological representations, in this case the variant between *infinite* and *vast*, in the layer `v1` and on the baseline `b0`. Secondly, all annotation that does not represent editorial operations was extracted from the baseline and integrated into annotation layers in their own right and are represented as layers `l1` through `l7`. This makes it possible to create a reduced display such as listing 4.29.

Once a graphical user interface is available where one can selectively hide and show information, it will be possible to dynamically reduce listing 4.28 to the more compact listing 4.29, and other similar stages.

The purpose of such dynamic reduction could be to support the current focus of a philologist's work, making focused interaction with the source a lot simpler.

Listing 4.29 Deep import of XML into codex (selective display)

```
.                  .       .                                                .
                   vast                                                    | v1: b0
...can show us the infinite multitude of the means of our language...      | b0
               [7 annotations hidden...]
```

4.4.2 Data conversion for "Wiener Ausgabe"

In the framework of the larger DH project I am part of, the first use case for *codex* is the representation of Wittgenstein's writing in the so-called Wiener Ausgabe, an example of which we have seen in chapter 2.

Consider listing 4.30, which shows an example remark from the Wiener Ausgabe in WittgenEd syntax, the custom markup language which is used to produce the print edition.

Listing 4.30 Example remark from the Wiener Ausgabe

```
$Bem
Die meisten Menschen, wenn sie eine philosophische Untersuchung
$Pag 46
anstellen $Esollen\E $K$V$Wbenehmen sich\W%%$Wsind wie\W%%machen es\V wie
$V$SEiner der …\S%%$Sein sehr nervöser Mensch …\S%% $KEiner der$K $-
a\-$+ä\+ußerst nervös\V $Sund in Hast …\S $*$-E\-e\*inen Gegenstand
in einer Lade sucht. Er $Kwirft Papiere aus der Lade heraus - das Gesuchte
$Vmag%%$S\/kann\/\S\V unter ihnen sein\K - blättert hastig und ungenau unter den
übrigen. $KWirft wieder die ersten $SPapiere\S zurück und $Vandere\K
heraus%%bringt sie mit den anderen durcheinander\V, u.s.w.. Man kann
diesem dann nur sagen: $KHalt, wenn Du $Uso\U suchst kann
$V$Sich nicht mit Dir …\S%%ich Dir nicht\V $Esuchen\E helfen. Erst mußt Du
anfangen in vollkommener
$Pag 47
Ruhe und methodisch eins nach dem andern zu untersuchen dann $Kbin ich auch
bereit Dir suchen zu helfen [mit Dir zu suchen] und $Sauch …\S mich auch in
meiner Methode nach Dir zu richten.
\Bem
```

The entire passage begins with a `$Bem$`-tag and closes with `\Bem` for "Bemerkung" ("remark"). Further tags are:

- `$E ... \E` for 'Einfügung', i.e. an insertion
- `$S ... \S` for 'Streichung', i.e. a deletion
- `$U ... \U` for 'Unterstreichung', i.e. underline
- `$W ... \W` for 'Wellige Unterstreichung', i.e. a wavy underline

Although the format mostly follows the classic syntax of opening and closing tags, it nevertheless has some peculiarities: The custom markup language adds `$V ... %% ... \V` for parallel, open variants. It is even possible to express more than two variants; this is done by stacking further variants by using more `%%`-seperators. It also uses `$K`-commands, which

set context flags, used to determine contexts for the automatically-prepared footnotes. A peculiarity of the `$K`-tags makes parsing difficult: they must be searched for in a backwards manner from a point where an actual operation happens. Note that there are empty element tags such as `$Pag 46` which indicates a page break and stands on a line of its own. This tag does not have a closing tag pair. Otherwise, a classic pairing of opening and closing tags is followed, where opening tags begin with a $-sign and closing tags begin with a \-sign.

We can see that despite the relative simplicity of this markup in comparison to other transcription systems, even this small example becomes tedious to follow.

Converted into *codex*, this same transcript corresponds to listing 4.31. The visual simplicity of the topological format is in striking contrast with the original markup.

There are nine blocks created by wrapping. As discussed before, each only has the necessary layers: the second block has 13 layers, whereas the last three have only one layer. There are seven deletion layers in this remark (d1-d7), two insertions (i1, i2) and seven variant layers (v1-v7). One of the variant situations even involves triple readings, namely the open variant of *sind wie*, *benehmen sich*, *machen es* (b0, v1, v2 in block 2). In my format, triple open variants are expressed by 2 variants layers.

From the annotation we can see that apparently Wittgenstein was dissatisfied with the first version, the baseline: he uses a wavy underline (layer a3) and writes variant v1. Again dissatisfied, he again underlines this variant with a wavy underline (a4) and writes variant v2. In most cases with this author, a wavy underline signals dissatisfaction with the given piece.

Listing 4.31 Example remark from the Wiener Ausgabe in codex notation; this listing has been cut into several blocks to make the visualisation here possible.

```
                                                                  .  .       .     .
                                                                                sollen | i1: b0
Die meisten Menschen, wenn sie eine philosophische Untersuchung anstellen       | b0
                                                                ^ Pag 46        | a1: b0
                                                                   ^ K          | a2: b0
```

```
.           .      ..        ..               .       .    .
                                              ------------- | d3: b0
                   Einer der äußerst nervös~~~              | v4: d2
                   -------------------------                | d2: v3
                   ein sehr nervöser Mensch …~              | v3: d1
                   -----------                              | d1: b0
machen es~~~~~~                                             | v2: b0
sind wie~~~~~~~                                             | v1: b0
benehmen sich__ wie Einer der …_________________ und in Hast … | b0
------------- W                                             | a3: b0
-------- W                                                  | a4: v1
                   ^ K                                      | a5: v4
                   ^ K                                      | a6: v4
                   - Emendation-Substitution: a             | a7: v4
```

```
..                .                    .                                .
einen Gegenstand in einer Lade sucht. Er wirft Papiere aus der Lade heraus - | b0
- Emendation-Substitution (insignificant): E                               | a8: b0
                                          ^ K                              | a9: b0
```

```
.           .          .            .                                                .
            --------                                                                 | d4: v5
            \/kann\/                                                                 | v5: b0
das Gesuchte mag_____ unter ihnen sein - blättert hastig und ungenau unter den übrigen. | b0
                                    ^ \K                                             | a10: b0
```

```
.                      .      .           .     .                                        .
                                          bringt sie mit den anderen durcheinander      | v6: b0
                       -------                                                          | d5: b0
Wirft wieder die ersten Papiere zurück und andere heraus_____________________________, u.s.w.. | b0
^ K                                                                                     | a11: b0
                                                 ^ \K                                   | a12: b0
```

```
.  .                            .             . .            .                  ..     .     . .
                                                                                 suchen         | i2: b0
                                                                  ich Dir nicht~~~~~~           | v7: d6
                                                                  --------------------          | d6: b0
Man kann diesem dann nur sagen: Halt, wenn Du so suchst kann ich nicht mit Dir …        helfen. | b0
                                ^ K                                                             | a13: b0
```

```
.                                                                                               .
Erst mußt Du anfangen in vollkommener Ruhe und methodisch eins nach dem andern zu untersuchen dann | b0
```

```
.                                                          .
dann bin ich auch bereit Dir suchen zu helfen [mit Dir zu suchen] | b0
```

```
.   .     .                                          .
    ------                                          | d7: b0
und auch … mich auch in meiner Methode nach Dir zu richten. | b0
```

The actual conversion from WittgenEd markup to the corresponding *codex* notation was done by an intern to the project, Marcel Eisterhues, using python, under my supervision. The source conversion proceeded as follows:

- Find all structural elements and record their positions, i.e. all opening and closing tags. However, self-closing tags, such as `$Pag` must be treated separately since they do not have a natural closing tag but instead end at the current line break.
- Pair opening and closing elements and merge them. Elements are classified as one of the editorial operations at this stage (e.g. `$V ... %% ... \V` is explicitely classified as an open variant at this stage)
- Handle special annotations, such as `$V ... %% ... \V`, which need splitting into several elements.
- Order elements according to their respective start and end positions.
- Assign `id`s and `pid`s. Id-prefixes are assigned to layers according to their type, e.g. `i1` for the first insertion, `v2` for the second variant.
- Remove opening and closing tags from the source and arrange elements vertically; this includes adding or removing space accordingly.
- Write the block out.

Only a small part of the entire Wiener Ausgabe data could be imported. This is because much of the data is still stuck on old storage tapes and could not be revived at this time. Note also that the majority of these documents are typescripts, meaning that they have been simplified from the handwritten sources. Nevertheless, the data conversion represents a proof-of-concept for *codex*' operation.

When we decided to use conversion rather than import as the method to input the Wiener Ausgabe into *codex*, the deciding factors were not only the peculiarities of the format, but also logistical reasons: at the time of the internship, my research was ongoing and the core routines of the *codex* library were not finished yet. Theoretically, an import on *codex*' library level instead of a format conversion should be possible, which would imply writing import routines to obey the peculiarities of the original format.

Finally, in table 4.1, I present some statistics about the data imported from the Wiener Ausgabe. Here, "rem" stands for remark, "ins" stands for insertions, "del" for deletions, "var" for open variant occurrences, "vars" for the actual number of variants (in case there are triple variants or more, as seen above), "math" for mathematical formulae, and "ops" for the combined sum of editorial operations. Files Ms-153–155 are manuscripts. Files Ts-208–218 are typoscripts; they were dictated by Wittgenstein.

Table 4.1: Statistics: Wiener Ausgabe data

file	words	rem	ins	del	var	math	vars	ops
Ms-153.wa	35067	622	121	833	402	460	419	1356
Ms-154.wa	16632	215	72	391	178	428	186	641
Ms-155.wa	14928	256	71	384	232	260	243	687
Ts-208.wa	54851	805	11	510	109	1345	110	630
Ts-210.wa	26872	375	2	444	156	505	158	602
Ts-211.wa	228994	2487	51	1233	1767	5443	1799	3051
Ts-212.wa	232465	2289	208	1416	1740	4773	1762	3364
Ts-213.wa	195423	2167	6		1216	1712	1228	1222
Ts-214.wa	3635	43			19	18	19	19
Ts-215.wa	5500	31			21	31	21	21
Ts-216.wa	1503	20			6	22	6	6
Ts-217.wa	1196	13			2	3	2	2
Ts-218.wa	739	10			8	2	8	8
Total	817805	9333	542	5211	5856	15002	5961	11609

Table 4.2: Relative frequency of editorial operations in WA (manuscripts)

file	ins/rem	del/rem	var/rem	ops/rem	math/rem
Ms-153.wa	0.19	1.34	0.65	2.18	0.74
Ms-154.wa	0.33	1.82	0.83	2.98	1.99
Ms-155.wa	0.28	1.50	0.91	2.68	1.02
Average	0.27	1.55	0.80	2.61	1.25

Table 4.3: Relative frequency of editorial operations in WA (typoscripts)

file	ins/rem	del/rem	var/rem	ops/rem	math/rem
Ts-208.wa	0.01	0.63	0.14	0.78	1.67
Ts-210.wa	0.01	1.18	0.42	1.61	1.35
Ts-211.wa	0.02	0.50	0.71	1.23	2.19
Ts-212.wa	0.09	0.62	0.76	1.47	2.09
Ts-213.wa	0.00	0.00	0.56	0.56	0.79
Ts-214.wa	0.00	0.00	0.44	0.44	0.42
Ts-215.wa	0.00	0.00	0.68	0.68	1.00
Ts-216.wa	0.00	0.00	0.30	0.30	1.10
Ts-217.wa	0.00	0.00	0.15	0.15	0.23
Ts-218.wa	0.00	0.00	0.80	0.80	0.20
Average	0.01	0.29	0.50	0.80	1.10

These parts of the Wiener Ausgabe in *codex* contains 9333 remarks, spanning 817805 words[6] This makes a remark in Wittgenstein's work 87 words long on average.

The total number of editorial operations recorded is 11609. Variants are the most frequent editorial operation (5856), with deletions following closely behind (5211). Insertions are an order of magnitude fewer, at 542 cases. In our example above, we saw a triple open variant. The difference between "var" and "vars" tells us how often variants go beyond the minimal case of two variants. 105 variant layers are involved in above-two open variants. So even in Wittgenstein's writing, such triple and more variants are rare[7].

One might not expect editorial operations in the typoscripts at all. Once his thoughts were typed up, Wittgenstein cut the typed pages back to remark-level, which have been carefully recorded by philologists (cf. the stemma given on page 24). After cutting, Wittgenstein immediately revised his own writing again. This is why we find editorial operations on these typed pages.

We also see how extensively Wittgenstein uses mathematical formulae in his arguments: 15002 times in my text sample, so on average more than once per remark. Ultimately, this is not surprising, since the text samples all follow along the genealogy of the stemma, i.e. there exists a core unit of remarks in different stages and at the different times, which Wittgenstein keeps revising.

We can also record the occurrence of the editorial operations per remark, as is shown in Tables 4.2 and 4.3, split into manuscripts and typoscripts.

The remarks are variants of each other, as the stemma shows (page 24). They do not create a single final text; this is not necessarily what Wittgenstein wanted and this is not how he operated. It is therefore not possible to determine from this data how many sensible linearisations there are in my text piece; it would need philological work to do so. What we can do with the corpus statistics, however, is to present some upper bounds on the number of linearisations theoretically (combinatorically) possible.

If we treat the remarks as if they constituted a coherent text, which we know they don't, our estimate of the number of linearisations would be

$$e_1 = \prod_{i=2}^{m} i^{p_i}$$

with m the maximum variant cardinality overall, i the index for cardinality of an editorial

[6]Remember that the remark is Wittgenstein's basic unit, which he keeps re-writing and re-arranging, as the stemma on page 24 shows.

[7]ops = ins+del+var, so ops records the instances of operations occurring. Both triple variants and double variants count as one.

operation, and p_i the number of editorial operations with cardinality i. (The cardinality is 2 for most editorial operations, and only higher for variants with more than 2 lines.)

This constitutes a very large upper bound of the number of linearisations. Another estimation is possible where we treat each remark as a stand-alone item, because we know that many remarks are variants of each other. In this case, we assume that editorial operations from one remark don't affect linearisations outside that remark itself. This brings us to the following estimate:

$$e_2 = \sum_{j=1}^{l} \prod_{i=2}^{m_j} i^{p_{i,j}}$$

possible linearisations per remark, with l the number of all remarks (and j the index for remarks), m_j the maximum cardinality of editorial operations in remark j, and $p_{i,j}$ the number of editorial operations of cardinality i in remark j.

These two estimates e_1 and e_2 are different types of upper bounds for the number of possible *virtual documents* dormant in this (small) corpus. The second estimate should result in a far lower number. Dividing e_2 by the number of remarks would give us an idea of the average number of linearisations per remark.

It is important to note that both estimates presented here are theoretical constructs relying only on combinatorics. They are meant to show how big the search space is in which the real philological work takes place. Given what we know about Wittgenstein's working practices, they are of course unrealistically high. The real work happens in cutting down this space to the meaningful realm.

In fact, the document model presented in this thesis makes it possible for philologists to programmatically break down the immense set of interpretations into their individual interpretation. Rather than be limited by a data model and its constraints, the data model should enable them to create and share individual interpretations effortlessly, without compromising other philologists' interpretations.

Chapter 5

Conclusion

5.1 Summary of Contributions

Born from a collaboration with the Digital Humanities started in 2010, this work presents *codex*, a universal document model beyond the tree paradigm, accompanied by a standoff, topological notation syntax. The tasks in its remit are the recording, editing, studying, dissemination and archiving of complex textual artefacts. I have shown in the preceding chapters that and why it is particularly well-suited for complex document requirements including overlapping structures and multiple hierarchies over parallel texts. The document model exemplified by *codex* is unambiguous and interoperable. Its human-centric syntax is inspired by music notation and makes sustainable long-term archiving of complex textual artefacts possible.

By using a standoff notation format, the document model is fully compatible with both linear plain text files as well as structured documents of any format, without compromising the semantics of the original document. Additionally, it enables version control, collaboration, and unfettered annotation of any type of source document. The work presented here therefore fills a central gap in document models beyond the tree paradigm.

The status quo of digital text editing builds on tree-structured documents with embedded markup. Reliance on the mono-hierarchical data structure of the ordered tree with standardised elements buys us the advantage that the compliance of the structure with a context-free grammar (called the document schema) can be validated. The tree-structured nature of this format however strictly limits it to one stream of linear text and non-overlapping markup. For a majority of use cases in commerce and science this is a viable choice: through the standardised markup, these documents are interoperable and can be exchanged freely between people, machines and projects.

However, when it comes to more complex requirements, the tree model is not powerful enough. Among these requirements are overlapping structures, non-contiguous structures, as well as multiple hierarchies. Many of the phenomena worth representing in a DH situation are of this kind and therefore cannot be described with a single-inheritance tree.

This has been realised in the DH community, and there have been repeated attempts to design a better document model for these more complex scenarios (from Nelson 1965 to D. Schmidt and Colomb 2009). However, in practice, what happens in most cases is that practitioners remain in the tree-model and pervert it so that it works for their data and use case. The necessary workarounds make the documents look like valid trees on syntactic level, so they can still be formally validated, although the validation has lost its meaning. Documents become idiosyncratic, introducing a brittleness that affects all downstream processing that cannot be avoided. By working around the tree paradigm, it is no longer possible to enjoy the advantages that come with the tree model: guaranteed well-formedness and validity with respect to a document schema, and use of many tools.

As a consequence, the resulting documents become a maintenance burden as they cannot be processed generically. The tree-structured document model is therefore an inadequate choice for complex document requirements such as the archiving and digital preservation of textual cultural heritage (D. Schmidt 2010).

Starting from the use case of the philological reconstruction of complex manuscripts as Digital Scholarly Editions, the solution presented in this thesis aimed to solve these problems, generically and for all document types and requirements. It does so by inventing a topological document model that goes beyond the tree model.

This work has shown how to archive concurrent hierarchies over concurrent text through a multi-dimensional matrix structure, inspired by music notation. I have argued that this enables both longevity and interoperability. By lifting the confinements of the tree paradigm, document object models which would otherwise necessarily be ambiguous are resolved and turned into unambiguous structures. By using a graph and not a tree for some aspects of my solution, I am no longer forced to work with the constraint that there can only be one single parent node for each text piece. This allowed me to change the context-free nature of data representation in XML into one that is context-sensitive.

My solution shifts the problem space up by one dimension. Information that would otherwise require verbose markup and machinery in a single dimension can be expressed simply through its topological arrangement in the topology of the matrix structure. This makes the syntax (the topological matrix notation) declarative, sparse and arguably elegant in its minimalism. Through its visual properties, it is naturally human-centric because it works with rather than against the cognitive system, while maintaining machine-readability.

Furthermore, through the topological and linear organisation of information, the collaborative editing of structured documents is greatly simplified. One could also track changes to the document using tree structures, but the complexity would grow combinatorially if we did that. In my solution we can add layers, but this can never explode the complexity because each layer can be treated independently.

The topological notation is simple and universal. It is applicable to everything that deals with ideas and their revisions, as long as the ideas can be expressed in symbols. This mainly includes text, but also music. A wide range of additional applications beyond digital scholarly editions are imaginable, from business documents requiring exact change-tracking, to collaborative online editing of structured documents, to source code versioning with semantic search.

As far as document models in computer science are concerned, this work presents a novel document model beyond the tree paradigm, extending context-free approaches to context-sensitive ones. Moreover, this work also contributes towards a philology-based theory of editorial operations, extending the classic diff-model.

In the following, I will pinpoint the contributions more specifically.

Polyglot, language-agnostic, standoff multi-markup By freeing the text from embedded markup and the restrictions imposed by the tree model, *codex* allows for any type of source document to be annotated and marked up, using any markup scheme. Moreover, it also allows to use more than one scheme at any one time, thus enabling multi-markup. For instance one might have a source that is marked-up using three different technologies or tag-sets: LaTeX, as well as markdown, as well as TEI (in XML). At any time, each of these tag-sets can be exported and rendered through the respective typesetter, validator or compiler.

Technology-independent long term archiving Future-proofing of the document model is achieved in that it is both human-intelligible as well as unambiguous and always truly machine-readable. The simplicity of the model makes the format technology-independent and therefore suitable for sustainable long-term archiving. Both the format as well as its underpinning implementation should easily be implementable on any future computer architectures.

Complexity reduction Another factor is complexity reduction: In this format it is extremely simple to suppress unnecessary information, just like one can do in video or music editing software by the use of different tracks. Since the layers can be treated independently or in groups, it is possible to filter them out in any desired representation.

Information about different aspects (such as descriptions of the appearance, material aspects of page size, structural annotation or commentary) is handled by different layers. In codex, data aspects that needn't mix don't mix. In this way, they do not interfere with one other and should significantly simplify data curation.

Novel methods for recording, editing, archiving and dissemination With tree models (XML in this case), automated extraction of any piece of information needs a directed walk from the source (root) to the desired point (node) of interest, informed by the schema. Such a schema can easily amount to ca. 600 tags, innumerable possible attributes and ca. 1500 pages of description, as is the case for TEI. All of these need to be taken into account at any given time, by program or human. (This complexity exists even if we don't take the nasty side-effects of the different workaround methods into account).

With the topological standoff markup approach of the multi-dimensional matrix, query and extraction allow for a completely different paradigm: instead of drilling down to the desired piece of information kept in a tree one can instead consult the desired piece of information by directly looking it up in the independent layers, cleanly and declaratively.

It is my belief that my approach has the ability to simplify complicated projects and operations by a large margin. Also, it changes the paradigm from a machine-oriented one, where philologists are forced to slavishly run after XML errors and thus cater to the machine, to a human-driven one, where any information the philologist wants to record is a priori assumed to be acceptable. The user should have nothing to do with the data model. They should only be asked for input when formal problems arise, so that they can make sure that the representation they create expresses what they want it to express. This aligns with *codex*'s syntax and overall philosophy in democratising access to valuable sources from the cultural heritage domain by simplifying methods and procedures for the general public.

5.2 Future work

This work has provided the design and a large part of the annotation of *codex*, but any PhD work by its nature is always incomplete and there are various avenues of future work. I will now propose three main applications for future work.

5.2.1 Graphical User Interface

My current implementation provides a commandline interface; a technically-minded philologist is probably already able to work with the commandline interface for a project, although it requires some getting used to. A future aim could be to create a graphical

user interface for curating texts, which should recreate the immediacy as pen and paper, and which should be suitable for the average scholar in the humanities.

In section 2.1, I talked about the advantages of pen on paper. Ideally, one would like to simulate exactly what a scholar can do with pen and paper, as much as this is possible in the electronic sphere. Such a simulation of course, has its limitations: it is well-documented from psycholinguistic experiments on reading that printed paper supports efficient reading far better than any electronic paper simulation does (O'hara and Sellen 1997; Dillon 1992). But what I am proposing here involves editing and curation work, not reading; I am not aware of any direct studies of such user interfaces.

Philological reconstruction of complex manuscripts requires a lot of focus and attention. Thus the editor should be kept free from any distracting information. The document model, through its visual aspect, already provides a good starting point in that it keeps the working environment free from distractions. A well-thought out GUI can boost this even further, namely by offering means and methods to suppress information that is not necessary at the current point in the philological process. The document model builds on layers and these layers could be suppressed just as in video or audio software, where one can hide tracks which are not of interest at the current moment. Moreover, by lifting the strict hierarchy of the tree model, annotation and data modelling is greatly simplified.

Secondly, a GUI should also update the variant graphs in the background and have them ready for inspection when this is requested. Instant access to linearisations of the current piece can aid the reconstruction process and reveal inconsistencies or mis-modelled pieces. The inheritance graph of the layers (the DOM) could also be displayed upon request to provide further information.

Since the biggest advantage of the printed book is the clarity of its proper typesetting, well-typeset exports can aid proof-reading. A desirable functionality would therefore be the export of data to different formats, combined with direct rendering (e.g. professional typesetting formats, HTML, LaTeX).

All in all, the GUI should provide a clear interface, tailor-made to the editor's needs for deep focus work on intricate problems, including hotkeys for starting insertions, switching perspectives and similar tasks. The design of the GUI will require more collaboration with the Digital Humanities to determine the exact needs of of editors and also involves experimentation. However, since the backend-logic is available through the data model presented here, frontend developers can find a good starting point for their experimental work.

5.2.2 Extension to music notation and other modalities

As music notation constitutes the founding idea for this thesis, I will now consider how music itself could be represented in the format I designed, as well as other modalities such as mathematical formulae and possibly figures. Just as in music notation, where several voices or instruments are synchronised with each other, the topological notation format also allows for more voices to come, join, and leave. In music, some voices or instruments can pause for a long time while the main line is active, and yet are easily visible through the voice's position in topological space. It follows that the identity of each voice remains both well-defined and intuitive. Because of the common underlying principles of the two notation systems, the idea to represent music instead of text seems a natural next step in expanding the reach of the format.

The Digital Humanities are not only occupied with textual sources, but also with musical sources. In this case, music instead of text is used as the language to express ideas. The process of composing music, just as that of writing text, is complex and creates information beyond the final musical notation, such as variants or fingering marks for piano. When musical works are created, composers frequently change their mind and rewrite parts. The reconstruction of the gestation of musical works is one of the tasks of musical philology, just as in textual philology. An example for such curation work is the Digital Variorum Edition (Rink et al. 2015), where the sources are reconstructed, the gestation of these works is carefully modeled, and different witnesses are collated.

The MEI, the Music Encoding Initiative's Guidelines, was created to reconstruct such musical sources. However, although the TEI and MEI share important aspects, they are still separate entities. Because of *codex*' polyglot nature, the topological notation format presented here could help merge both of these projects together.

Mathematics Similarly, mathematics can be seen as a language in its own right, which should be integrated properly with the data model. For example, Wittgenstein often makes use of mathematical formulae in his philosophical arguments.

Representing maths in the DH world hasn't received much attention, but some DH projects use TEX, which provides excellent typesetting. But TEX is a macro language designed only for the typesetting of formulae and not for their declarative and semantic representation, as would be necessary for unambiguous interpretation and processing. If we had a semantic format, we could query and index, even perform sub-formulae matching from Mathematical Information Retrieval (MIR). MIR is a thriving research area with common shared tasks and a large community of practicioners (e.g., Aizawa and Kohlhase 2021; Mansouri et al. 2019). Related to this is the work on MathML, the commonly used standard for all kinds

of processing of the mathematical discourse.

Some extra work would have to be done to integrate formulae at this semantic level into the data model. The first step would be to integrate MathML with editorial operations, as it would then be possible to express the gestation of formulae and how they are linguistically connected to the text. This is necessary because the formulae themselves (or parts of these), as well as their connections to the text, are subject to corrections. In any such extension, it is a good idea to start with a thorough analysis of philologists' needs for mathematical representation in real DH settings.

Figures Wittgenstein uses figures extensively to explain philosophical concepts. For instance, he uses the illustration of a clutch from automotive engineering to explain a philosophical concept.

As with mathematical formulae, to implement a workable representation of figures, a good description language is necessary. Like mathematical formulae, such figures would need typesetting and, ideally, indexing and search. Ideas and concepts are of course linguistically expressed, but they can be enriched with figurative language (e.g. metaphors), and even actual drawings and figures. Adequately exposing the semantic content of such figures and connecting it back to the linguistic expressions could constitute a new avenue of research in philology and computational linguistics.

An initial step in this direction might be simple search mechanisms which might be described as "schematic" search. There are already approaches which allow the user to search for shapes by sketching the desired shape (Xi et al. 2007; Zhang and Lu 2004) or symbol (Kirsch 2010). Another approach could allow search via language description (with queries like "circle around dot within a rectangle on the right").

I have discussed the differences between collaborative editing and philology in section 2.6. Be that as it may, there still is a large amount of overlap between the fields of philology and collaborative editing. With this work, I hope to spark innovation in the form of novel tools for both fields, in the spirit of Sahle's (Sahle 2014) quote at the beginning of this thesis (p. 12).

With this thought I would like to conclude my thesis. I have presented a solution that I believe to be adequate to the needs in philology while at the same time offering solutions to long-running problems in computer science: the inability to model non-linear text and support markup beyond the tree paradigm. Computer science has much to gain by opening itself up to centuries of philological expertise, which inspired a radically different text model and with it, new ways of multi-hierarchical markup and concurrent annotation. It is my hope for the future that my document model and the ideas presented with it

will enable improved versions of existing tools like change-tracking in business documents, collaborative editing of structured documents, or altogether novel tools, like semantic source code versioning, structural code search, robust natural language parsing, music notation as well as language-agnostic annotation and federated data curation of any source.

Bibliography

Abouelaoualim, Abdelfattah, Valentin Borozan, Yannis Manoussakis, Carlos AJ Martinhon, Rahul Muthu, and Rachid Saad. "Colored Trees in Edge-Colored Graphs." In *CTW*, 115–119. 2009.

Aizawa, Akiko, and Michael Kohlhase. "Mathematical information retrieval." *Evaluating Information Retrieval and Access Tasks: NTCIR's Legacy of Research Impact*, 2021, 169–185.

Barabucci, Gioele. "A universal delta model," 2013.

Barabucci, Gioele, Paolo Ciancarini, Angelo Di Iorio, and Fabio Vitali. "Measuring the domain-oriented quality of diff algorithms." *delta* 9 (2016): 6.

———. "Measuring the quality of diff algorithms: a formalization." *Computer Standards & Interfaces*, 2016.

———. "Towards a Qualitative Analysis of Diff Algorithms." In *IIR*, 33–36. 2013.

Barabucci, Gioele, Angelo Di Iorio, Silvio Peroni, Francesco Poggi, and Fabio Vitali. "Annotations with EARMARK in practice: a fairy tale." In *Proceedings of the 1st International Workshop on Collaborative Annotations in Shared Environment: metadata, vocabularies and techniques in the Digital Humanities*, 11. 2013.

Barabucci, Gioele, Silvio Peroni, Francesco Poggi, and Fabio Vitali. "Embedding semantic annotations within texts: the FRETTA approach." In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 658–663. 2012.

Bazzocchi, Luciano. "A Better Appraisal of Wittgenstein's Tractatus Manuscript." *Philosophical Investigations* 38, no. 4 (2015): 333–359.

Berners-Lee, Tim, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. "Tabulator: Exploring and analyzing linked data on the semantic web." In *Proceedings of the 3rd international semantic web user interaction workshop*. Vol. 2006. 2006.

Bikakis, Nikos, Melina Skourla, and George Papastefanatos. “rdf: SynopsViz–a framework for hierarchical linked data visual exploration and analysis.” In *European Semantic Web Conference*, 292–297. 2014.

Bizer, Christian. “The emerging web of linked data.” *Intelligent Systems, IEEE* 24, no. 5 (2009): 87–92.

Bleeker, Elli, Bram Buitendijk, and Ronald Haentjens Dekker. “Marking up microrevisions with major implications: Non-linear text in TAG.” In. Mulberry Technologies, Inc. 2020. https://doi.org/10.4242/balisagevol25.bleeker01. 10.4242/balisagevol25.bleeker01.

Bleeker, Elli, Ronald Haentjens Dekker, and Bram Buitendijk. “Hyper, Multi, or Single? Thinking about Text in Graphs and Trees.” In. Mulberry Technologies, Inc. 2021. https://doi.org/10.4242/balisagevol26.bleeker01. 10.4242/balisagevol26.bleeker01.

Bruder, Daniel. “Symmetric Index Structures – Highly efficient symmetric text indexing.” Master’s Thesis, LMU Munich, 2012. http://www.cip.ifi.lmu.de/~bruder/ma/MA/sis/MA-SIS-DB.pdf.

Chiarcos, Christian. “Interoperability of corpora and annotations.” In *Linked Data in Linguistics*, 161–179. Springer, 2012.

Chiarcos, Christian, Sebastian Hellmann, and Sebastian Nordhoff. “Linking linguistic resources: Examples from the open linguistics working group.” In *Linked Data in Linguistics*, 201–216. Springer, 2012.

———. “Towards a Linguistic Linked Open Data cloud: The Open Linguistics Working Group.” *TAL* 52, no. 3 (2011): 245–275.

Chiarcos, Christian, John McCrae, Philipp Cimiano, and Christiane Fellbaum. “Towards open data for linguistics: Linguistic linked data.” In *New Trends of Research in Ontologies and Lexical Resources*, 7–25. Springer, 2013.

Consortium, TEI. *A Gentle Introduction to XML*. Chap. v in *TEI P5: Guidelines for Electronic Text Encoding and Interchange*, by TEI Consortium, Version 3.2.0. 2016. Accessed June 1, 2017. %7Bhttp://www.tei-c.org/release/doc/tei-p5-doc/en/html/SG.html#SG152%7D.

———. *Non-Hierarchical Structures*. Chap. 20 in *TEI P5: Guidelines for Electronic Text Encoding and Interchange*, by TEI Consortium, Version 3.2.0. 2016. Accessed June 1, 2017. http://www.tei-c.org/release/doc/tei-p5-doc/en/html/NH.html.

Dekker, Ronald Haentjens, Elli Bleeker, Bram Buitendijk, Astrid Kulsdom, and David J. Birnbaum. “TAGML: A markup language of many dimensions.” In. Mulberry Technologies, Inc. 2018. https://doi.org/10.4242/balisagevol21.haentjensdekker01. 10.4242/balisagevol21.haentjensdekker01.

Dekker, Ronald Haentjens, Bram Buitendijk, and Elli Bleeker. “Parsing a markup language that supports overlap and discontinuity.” In. ACM, 2020. https://doi.org/10.1145/3395027.3419590. 10.1145/3395027.3419590.

DeRose, Steve, Eve Maler, and Ron Daniel. “XML pointer language (XPointer),” 2000.

Di Iorio, Angelo, Silvio Peroni, and Fabio Vitali. “A Semantic Web approach to everyday overlapping markup.” *Journal of the American Society for Information Science and Technology* 62, no. 9 (2011): 1696–1716.

———. “Handling markup overlaps using OWL.” In *Knowledge Engineering and Management by the Masses*, 391–400. Springer, 2010.

Dillon, Andrew. “Reading from paper versus screens: A critical review of the empirical literature.” *Ergonomics* 35, no. 10 (1992): 1297–1326.

Erbacher, Christian. “Editorial Approaches to Wittgenstein’s Nachlass: Towards a Historical Appreciation.” *Philosophical Investigations* 38, no. 3 (2015): 165–198.

———. “Unser Denken bleibt gefragt: Web 3.0 und Wittgensteins Nachlass.” *Wissenschaftstheorie, Sprachkritik Und Wittgenstein. Heusenstamm: Ontos*, 2011, 135–146.

Falch, Rune, Christian Erbacher, and Alois Pichler. “Some observations on developments towards the Semantic Web for Wittgenstein scholarship.” In *36th International Ludwig Wittgenstein Symposium*, 11–17. 2013.

Falch, Rune J, Heinz Wilhelm Krüger, and Deirdre Smith. “Elements of an e-Platform for Wittgenstein Research.” *From the WAB archives: A selection from the Bergen Wittgenstein archives Working Papers and audio-visual materials*, 2012.

Fanta, Walter. “Die Computer-Edition des Musil-Nachlasses.” *editio* 8 (1994): 127–158.

———. *The Genesis of Der Mann ohne Eigenschaften*. In *A Companion to the Works of Robert Musil*, NED - New edition, 251–284. Boydell & Brewer, 2007. http://www.jstor.org/stable/10.7722/j.ctt169wdm1.15.

Ferré, Sébastien. “Conceptual navigation in RDF graphs with SPARQL-like queries.” In *International Conference on Formal Concept Analysis*, 193–208. 2010.

Ferrié, Jean, Nicolas Vidot, and Michelle Cart. “Concurrent undo operations in collaborative environments using operational transformation.” In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, by RobertMeersman ZahirTari WilvanderAalst, ChristophBussler AvigdorGal VinnyCahill, SteveVinoski WernerVogels TizianaCatarci, and Katia Sycara. Springer, 2004.

Gerdjikov, Stefan, Stoyan Mihov, Petar Mitankin, and Klaus U Schulz. “Good parts first – a new algorithm for approximate search in lexica and string databases.” *arXiv preprint arXiv:1301.0722*, 2013.

Gerdjikov, Stefan, Stoyan Mihov, Petar Mitankin, and Klaus U Schulz. "Wallbreaker: Overcoming the wall effect in similarity search." In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 366–369. 2013.

Gradmann, Stefan. "Towards a Social Semantic Scholarly Graph: the Wittgenstein Incubatoras part of an attempt to further model scholarly discoursive interaction in DM2E," 2013.

Hadersbeck, Max, Alois Pichler, Florian Fink, and Øyvind Liland Gjesdal. "Wittgenstein's Nachlass: WiTTFind and Wittgenstein advanced search tools (WAST)." In *Proceedings of the First International Conference on Digital Access to Textual Cultural Heritage (DATeCH)*, 91–96. 2014.

Haentjens Dekker, Ronald, and David J. Birnbaum. "It's more than just overlap: Text As Graph." *Proceedings of Balisage: The Markup Conference 2017*, 2017. http://dx.doi.org/10.4242/BalisageVol19.Dekker01. 10.4242/balisagevol19.dekker01.

Hanrahan, Elise. "Over-tagging with XML in Digital Scholarly Editions." In *DHd2015 Conference – Von Daten zu Erkenntnissen. Book of Abstracts.*, edited by Various, 162–165. Graz, Austria, 2015.

Haslhofer, Bernhard, and Antoine Isaac. "data. europeana. eu: The europeana linked open data pilot." In *International Conference on Dublin Core and Metadata Applications*, 94–104. 2011.

Haslhofer, Bernhard, Robert Simon, Robert Sanderson, and Herbert Van de Sompel. "The open annotation collaboration (OAC) model." In *Multimedia on the Web (MMWeb), 2011 Workshop on*, 5–9. 2011.

Horrocks, Ian. "Ontologies and the semantic web." *Communications of the ACM* 51, no. 12 (2008): 58–67.

Hrachovec, Herbert. "Evaluating the Bergen Electronic Edition." *Wittgenstein: The philosopher and his works.*, 2005, 364–376.

———. "Wittgenstein on line/on the line." *The Wittgenstein Archives at the University of Bergen (WAB)*, 2000. http://www.wittgensteinrepository.net/ojs/index.php/agora-wab/article/view/3241.

———. "Wittgenstein's Paperwork. An Example from the "Big Typescript"." 2006. http://wab.uib.no/agora/tools/alws/collection-6-issue-1-article-23.annotate.

Huitfeldt, Claus. "Multi-dimensional texts in a one-dimensional medium." *Computers and the Humanities* 28, nos. 4-5 (1994): 235–241.

———. *Toward a Machine-Readable Version of Wittgenstein's Nachlaß.* Edited by Hans Gerhard Senger. In *Philosophische Editionen: Erwartungen an sie - Wirkungen durch sie.*

Beiträge zur VI. Internationalen Fachtagung der Arbeitsgemeinschaft philosophischer Editionen (11.-13. Juni 1992 in Berlin), 37–43. Max Niemeyer Verlag, 2011. https://doi.org/10.1515/9783110939323.37. doi:10.1515/9783110939323.37.

Huitfeldt, Claus, and CM Sperberg-McQueen. "TexMECS: An experimental markup meta-language for complex documents." 2001. http://www.hit.uib.no/claus/mlcd/papers/texmecs.html.

Hunter, Jane, Tim Cole, Robert Sanderson, and Herbert Van de Sompel. "The open annotation collaboration: A data model to support sharing and interoperability of scholarly annotations." In *Digital humanities 2010*, 175–178. 2010.

Kirsch, Daniel. "Detexify: Erkennung handgemalter latex-symbole." *German. Diploma thesis. Westfälische Wilhelms-Universität Münster*, 2010.

Kleppmann, Martin, and Alastair Beresford. "A Conflict-Free Replicated JSON Datatype." *IEEE Transactions on Parallel and Distributed Systems*, 2017.

Kuczera, Andreas. "Digital Editions beyond XML – Graph-based Digital Editions." Edited by Johannes Preiser-Kappeller Marten Düring Adam Jatowt. In *Proceedings of the 3rd HistoInformatics Workshop on Computational History (HistoInformatics 2016)*. 2016.

Maler, Eve, and Steve DeRose. "XML Pointer Language (XPointer). World Wide Web Consortium Working Draft," 2005.

Mansouri, Behrooz, Shaurya Rohatgi, Douglas W Oard, Jian Wu, C Lee Giles, and Richard Zanibbi. "Tangent-CFT: An embedding model for mathematical formulas." In *Proceedings of the 2019 ACM SIGIR international conference on theory of information retrieval*, 11–18. 2019.

Marcoux, Yves, Michael Sperberg-McQueen, and Claus Huitfeldt. "Modeling overlapping structures. Graphs and serializability." In *Balisage: The Markup Conference*, 6–9. 2013.

Max Hadersbeck et al. "Wittgenstein Source Bergen Nachlass Edition Transcription." Edited by Alois Pichler. Centrum für Informations- und Sprachverarbeitung (CIS), Ludwig-Maximilians-Universität München (LMU Munich) in cooperation with Wittgenstein Archives Bergen, Norway, 2020. Accessed February 20, 2020. http://reader.wittfind.cis.lmu.de/reader/Ms-115,129[2].

Miller, Robert C. "Lightweight Structure in Text." Published as CMU Computer Science technical report CMU-CS-02-134 and CMU Human-Computer Interaction Institute technical report CMU-HCII-02-103. PhD diss., Computer Science Department, School of Computer Science, Carnegie Mellon University, May 1, 2002.

Nedo, Michael. "Einführung/Introduction." In. Vols. Introduction of. Springer, 1993.

Nelson, Theodor H. “Complex information processing: a file structure for the complex, the changing and the indeterminate.” In *Proceedings of the 1965 20th national conference*, 84–100. 1965.

O’hara, Kenton, and Abigail Sellen. “A comparison of reading paper and on-line documents.” In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, 335–342. 1997.

Passant, Alexandre, and Philippe Laublet. “Meaning Of A Tag: A collaborative approach to bridge the gap between tagging and Linked Data.” In *LDOW*. 2008.

Peroni, Silvio, Aldo Gangemi, and Fabio Vitali. “Dealing with markup semantics.” In *Proceedings of the 7th International Conference on Semantic Systems*, 111–118. 2011.

Peroni, Silvio, and Fabio Vitali. “Annotations with EARMARK for arbitrary, overlapping and out-of order markup.” In *Proceedings of the 9th ACM symposium on Document engineering*, 171–180. 2009.

Peroni, Silvio, Fabio Vitali, and Angelo Di Iorio. “Towards markup support for full GODDAGs and beyond: the EARMARK approach,” 2009. 10.4242/BalisageVol3.Peroni01.

Pichler, Alois. “Advantages of a machine-readable version of Wittgenstein’s Nachlass: Beiträge des 18. Internationalen Wittgenstein Symposiums.” Edited by K. S. Johannessen and T. Nordenstam. *Culture and Value*, 1995, 770–776.

———. “Encoding Wittgenstein. Some remarks on Wittgenstein’s Nachlass, the Bergen Electronic Edition, and future electronic publishing and networking.” *TRANS. Internet-Zeitschrift für Kulturwissenschaften* 10 (2002).

———. *The Interpretation of the Philosophical Investigations: Style, Therapy, Nachlass.* Wiley Online Library, 2007.

———. *Towards the New Bergen Electronic Edition.* Springer, 2010.

———. “Towards Wittgenstein on the Semantic Web.” *Digital Humanities Conference 2012, University of Hamburg*, 2012.

Pichler, Alois, Deirdre Smith, Rune J Falch, and Wilhelm Krüger. Elements of an e-Platform for Wittgenstein Research.

Pichler, Alois, and Amélie Zöllner-Weber. “Sharing and debating Wittgenstein by using an ontology.” *Literary and linguistic computing* 28, no. 4 (2013): 700–707.

Rink, John Scott, John Scott Rink, Christophe Grabowski, and Christophe Grabowski. *Chopin Online*. University of Cambridge, 2015.

Rothhaupt, J. “Kreation und Komposition: Philologisch-philosophische Studien zu Wittgensteins Nachlass (1929–1933).” *Habilitationsarbeit. Ludwig-Maximilians-Universität München, München*, 2008.

Sahle, Patrick. “DH? Gibt’s doch gar nicht?! - Integration oder Desintegration der Digital Humanities in Deutschland.” Edited by Malte Rehbein and Patrick Helling. In *1. Tagung des Verbands Digital Humanities im deutschsprachigen Raum, DHd 2014, Passau, Germany, March 25 - 28, 2014*. 2014. https://doi.org/10.5281/zenodo.4623601. 10.5281/zenodo.4623601.

Sanderson, Robert, Paolo Ciccarese, and Herbert Van de Sompel. “Designing the W3C open annotation data model.” In *Proceedings of the 5th Annual ACM Web Science Conference*, 366–375. 2013.

Schmidt, Desmond. “The inadequacy of embedded markup for cultural heritage texts.” *Literary and Linguistic Computing* 25, no. 3 (2010): 337–356.

———. “The role of markup in the digital humanities.” *Historical Social Research/Historische Sozialforschung*, 2012, 125–146.

———. “Towards an interoperable digital scholarly edition.” *Journal of the Text Encoding Initiative*, no. 7 (2014).

Schmidt, Desmond, and Robert Colomb. “A data structure for representing multi-version texts online.” *International Journal of Human-Computer Studies* 67, no. 6 (2009): 497–514.

Schmidt, Thomas, Christian Chiarcos, Timm Lehmberg, Georg Rehm, Andreas Witt, and Erhard Hinrichs. “Avoiding data graveyards: From heterogeneous data collected in multiple research projects to sustainable linguistic resources.” In *6th E-MELD workshop, Ypsilanti*. 2006.

Schnitzler, Arthur. “Marionetten, ed. by Annja Neumann with Gregor Babelotzky, Judith Beniston, Julia Glunk, Kaltërina Latifi, Robert Vilain and Andrew Webber.” *Arthur Schnitzler digital, Historisch-kritische Edition (Werke 1905–1931)*, 2018, 35.

Seaborne, Andy, and Eric Prud’hommeaux. *SPARQL Query Language for RDF*. Http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. January 2008.

Shapiro, Marc, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. “A comprehensive study of convergent and commutative replicated data types.” Inria–Centre Paris-Rocquencourt; INRIA, 2011.

———. “Conflict-free replicated data types.” In *Symposium on Self-Stabilizing Systems*, 386–400. 2011.

Sperberg-McQueen, C Michael, and Claus Huitfeldt. “GODDAG: A data structure for overlapping hierarchies.” In *Digital documents: Systems and principles*, 139–160. Springer, 2000.

Sperberg-McQueen, CM, and Claus Huitfeldt. “Markup Discontinued: Discontinuity in TexMecs, Goddag structures, and rabbit/duck grammars.” In *Proceedings of Balisage: The Markup Conference*. Vol. 1. 2008.

Stern, David. “The Bergen Electronic Edition of Wittgenstein’s Nachlass.” *European Journal of Philosophy* 18, no. 3 (2010): 455–467.

Sun, Chengzheng, and Clarence Ellis. “Operational transformation in real-time group editors: issues, algorithms, and achievements.” In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, 59–68. 1998.

TEI Consortium. “History.” 2016. https://tei-c.org/about/history/.

———. *TEI P5: Guidelines for Electronic Text Encoding and Interchange*. Version 3.2.0. 2016.

Vitali, Fabio, Federico Folli, and Claudio Tasso. “Two implementations of XPointer.” In *Proceedings of the thirteenth ACM conference on Hypertext and hypermedia*, 145–146. 2002.

Waldron, Benjamin, and Ann Copestake. “A standoff annotation interface between DELPH-IN components.” In *Proceedings of the 5^{th} Workshop on NLP and XML: Multi-Dimensional Markup in Natural Language Processing*, 97–100. 2006.

Wang, Jinpeng, Jianjiang Lu, Yafei Zhang, Zhuang Miao, and Bo Zhou. “Integrating heterogeneous data source using ontology.” *Journal of Software* 4, no. 8 (2009): 843–850.

Wilde, Erik, and David Lowe. *XPath, XLink, XPointer, and XML: A practical guide to Web hyperlinking and transclusion*. Addison-Wesley Longman Publishing Co., Inc. 2002.

Wittgenstein, Ludwig, and Michael Nedo. *Wiener Ausgabe*. Edited by Michael Nedo. Springer, 1993–2023.

Xi, Xiaopeng, Eamonn Keogh, Li Wei, and Agenor Mafra-Neto. “Finding motifs in a database of shapes.” In *Proceedings of the 2007 SIAM international conference on data mining*, 249–260. 2007.

Zaytsev, Vadim. “Coupled Transformations of Shared Packed Parse Forests.” In *GCM@ ICGT*, 2–17. 2015.

Zhang, Dengsheng, and Guojun Lu. “Review of shape representation and description techniques.” *Pattern recognition* 37, no. 1 (2004): 1–19.

Appendix A

XML Transcript

The following transcription is taken from Max Hadersbeck et al. 2020. It shows the equivalent transcription to the text piece from the manuscript example of the introductory chapter.

```
<ab ana="field:PhilosophyOfLanguage_pub:W-EPB_date:19360827?-19361130?"
    emph="blbef_1" n="Ms-115,129[2]" xml:id="Ms-115_129.2" xml:lang="de">
  <seg type="wabmarks-secml"></seg>
  <seg type="publ">
    <s type="es">
      <choice type="em">
        <orig type="em1">
          <del type="d_c">Solche</del> Überlegungen
          <del type="d">
            <add rend="im">wie diese</add>
          </del>
        </orig>
        <orig type="em2">
          <choice type="dsf">
            <orig type="alt1">
              <c type="c">S</c>olche Überlegungen
            </orig>
            <orig type="alt2">Überlegungen wie diese</orig>
          </choice>
        </orig>
      </choice> können uns die
      <choice type="em">
        <orig type="em1">un
          <lb rend="shyphen"/>
          <del type="d">endliche</del>
          <add rend="i">geheure</add>
        </orig>
```

```
  <orig type="em2">
    <choice type="dsl">
      <orig type="alt1">unendliche</orig>
      <orig type="alt2">ungeheure</orig>
    </choice>
  </orig>
</choice> Man
<corr type="tran">n</corr>igfaltigkeit der Mittel unserer
<lb/> Sprache
<choice type="dsl">
  <orig type="alt1">
    <del type="d">zeigen</del>
  </orig>
  <orig type="alt2">
    <add rend="i">ahnen lassen</add>
  </orig>
</choice>; &amp;amp; es ist
<choice type="s">
  <orig type="alt1">merkwürdig, mit</orig>
  <orig type="alt2">
    <add rend="i">interessant mit</add>
  </orig>
</choice> dem
<lb/> was
<choice type="dsl">
  <orig type="alt1">
    <del type="d">wir hier
      <choice type="s">
        <orig type="alt1">sehen</orig>
        <orig type="alt2">
          <add rend="i">beobachten</add>
        </orig>
      </choice>
    </del>
  </orig>
  <orig type="alt2">
    <add rend="i">sich uns hier zeigt</add>
  </orig>
</choice>
<choice type="em">
  <orig type="em1">
    <del type="d">die einfachen &amp;amp; starren Regeln</del>
    <lb/> zu vergleichen,
    <del type="d">die</del>
    <add rend="i">was</add>
  </orig>
```

```
        <orig type="em2">
          <choice type="dsl">
            <orig type="alt1">die einfachen &amp;amp; starren Regeln zu vergleichen, die</orig>
            <orig type="alt2">zu vergleichen, was</orig>
          </choice>
        </orig>
      </choice> Logiker vom Bau aller Sätze
      <lb/> gesagt haben.
    </s>
    <s type="es">(
      <choice type="s">
        <orig type="alt1">Vergleiche auch, was ich</orig>
        <orig type="alt2">
          <add rend="i">
            <c type="c">D</c>ies gilt auch von dem, was ich
            <seg type="edinst"> …</seg>
          </add>
        </orig>
      </choice>
      <add rend="im">selbst</add> in
      <rs key="Wittgenstein, Ludwig: Logisch-philosophische Abhandlung"
          n="1921" type="extref" xml:lang="de">
        <abbr corresp="der Logisch-philosophischen Abhandlung">Log. Phil. Abh
          <corr type="tra">.</corr>
        </abbr>
      </rs>
      <lb/>
      <choice type="s">
        <orig type="alt1">gesagt</orig>
        <orig type="alt2">
          <add rend="i">geschrieben</add>
        </orig>
      </choice>
 habe.)
    </s>
  </seg>
</ab>
```

Listing 2: Introductory example (taken from Max Hadersbeck et al. 2020)