Schriften des Instituts für Dokumentologie und Editorik — Band 13

Versioning Cultural Objects Digital Approaches

edited by

Roman Bleier, Sean M. Winslow

2019

BoD, Norderstedt

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de/ abrufbar.

2019

Herstellung und Verlag: Books on Demand GmbH, Norderstedt
ISBN: 978-3-7504-2702-0
Einbandgestaltung: Julia Sorouri and Stefan Dumont; Coverbild gestaltet von Vinayak Das Gupta.
Satz: Roman Bleier, Sean M. Winslow und LuaTEX

The CMV+P Document Model, Linear Version[1]

Gioele Barabucci

Abstract

Digital documents are peculiar in that they are different things at the same time. For example, an HTML document is a series of Unicode codepoints, but also a tree-like structure, as well as a rendered image in a browser window and a series of bits stored on a physical medium. These multiple identities of digital documents not only make it difficult to discuss the evolution of documents (especially digital-born documents) in rigorous scholarly terms, it also creates practical problems for computer-based comparison tools and algorithms.

The CMV+P model addresses this problem providing a sound formalization of what a document is and how its many identities can coexist at the same time. In its linear version, described in this paper, the CMV+P model sees each document as a stack of *abstraction levels*, each composed of a) an addressable *Content*, b) a *Model* according to which the content has been recorded, and c) a set of *Variants* used for equivalence matching. The bottom of this stack is the *Physical* level, symbolizing the concrete medium that embodies the digital document. Content is moved across levels using *transformation functions*, i.e. encoding functions used to serialize (save) the document and decoding functions used to deserialize (read) it.

A practical application of the CMV+P model is its use in comparison tools, algorithms, and methods. With a clear understanding of the internal stratification of formats and models found in digital documents, comparison tools are able to focus on the most meaningful abstraction levels, providing the user with the ability to understand which comparisons are possible between two arbitrary documents.

1 Introduction

Finding differences and similarities between digital documents is fundamental for the study of digital cultural artifacts, as well as for their versioning and their preservation. With digital documents we mean both born-digital documents as well as proxy digital documents that represent other physical documents. Detecting differences between digital documents is, however, a complex task, not only because of the inherent algorithmic difficulties, but also because digital documents are stored in many different ways, using different formats and different models. For example, texts

[1] Received March 2017, published December 2019

could be stored as OpenDocument files (OpenOffice), PDF files, plain-text files, Google Docs, scanned printouts, and so on.

In addition to this plethora of digital document formats, there is another complication: the *stratification* of these formats. Each document exists, at least, at two different levels: the physical level (how it has been stored on a physical carrier) and the binary level (the logical sequence of zeros and ones it is composed of). In fact, common document formats employ many more levels of abstraction on top of the binary level; for example, an XML file can be seen, at same time, as a set of XML structures, as series of characters, or a string of binary digits.

The fact that the same string of bits represents at the same time multiple views on the same document often confuses users and scholars that study documents, especially those that study how these documents have been changed over time.

This confusion extends to comparison tools as well. Tools that find and describe the differences (or the similarities) between documents are based on algorithms that focus on only one of these many possible levels: e.g., only on the binary representation or only on the XML structures. For this reason comparison tools often produce unexpected and unusable results.

Take the example of an OpenOffice document that has been converted into a Microsoft Word file: while both files contain the same content, a comparison tool will say that these two files are completely different. This is paradoxical: how can two files with the same content be completely different?

A similar "equal but different" paradox arises when we compress files. For instance, an HTML page and a copy of it that has been compressed with gzip (Gailly and Adler). We know that both files have the same content, yet comparison tools will tell us that they are 100% different. How is this possible?

The root cause of these paradoxes is the lack of a precise and formal way to describe and refer to the stratification of abstraction levels that is present in every digital document.

Without the ability to understand this stratification, comparison tools will myopically see documents at one abstraction level only, often not the one the user is interested in. The lack of such a formalization makes comparison tools also unable to compare similar pieces of information (e.g., textual content) that have been stored using different formats (e.g., ODT vs DOC).

Connected to the stratification of documents, there are not only practical issues like those just described, but also epistemological problems. Without a clear understanding of the stratification of formats and models that occurs within digital documents, it is not possible to give precise and useful definitions of key concepts such as *version*, *revision*, *difference*, *change*, or even *document*.

This paper presents the CMV+P model (Content, Model, Variants + Physical embodiment), the aim of which is to provide a rigorous, formal, precise, and actionable way to identify and address the various levels of abstraction that exist in digital documents. Using CMV+P, humans and computers can state with precision at which level of abstraction they are performing their analysis. In the case of comparison tools, this means being able to describe which differences have been detected, on which parts of the document and at which abstraction level. Moreover, CMV+P makes it possible to meaningfully compare documents in different formats. In fact, CMV+P is a refined replacement for the document model originally designed for the Universal Delta Model (Barabucci, "Introduction"). Last, CMV+P enables scholars to reason about relations between different versions of the same document or about the evolution of documents which have changed in format or model over time.

The focus of this paper is the abstract description of the CMV+P model in its linear version. Future publications will describe more complex versions of this model for structured documents and present practical implementations.

2 How documents are written and read: an example

Before delving into the description of the CMV+P model, we should briefly discuss how digital documents (which we will refer to as simply *documents* from now on) are written (*serialized*) and read (*deserialized*). As a running example through this section and the rest of this paper, we will use a simple document that contains just the name of a fictive business: "Böh & Son."

In order to go from the concept of "a business name" to a series of bits, we will need to decide how to encode this abstract concept—or some kind of associated data structure—into a series of bits. The technical name for this process is *serialization*. As we will soon see, serializing a document consists of deciding how to describe an abstract piece of information into a less abstract piece of information. This is an iterative process: at each step we will deal with one class of details and will have to choose between multiple possibilities, all valid but with different associated trade-offs.

Media The very first choice we face is choosing which kind of media we want to use to record this name. We could make an audio recording while we pronounce the name of the business, we could draw that name (or the associated logo), or we could record it as "text."[2] In our case we will record this business name as text.

Text format Text can be stored digitally in many different ways, from *plain text* (Freed and Borenstein), where only text and no stylistic info is recorded, to more elaborate formats such as XML (Bray et al.), ODT (ISO 26300-1:2015), or PDF (ISO 32000-1:2008). To keep our example manageable we will use simple plain text.

Writing system Choosing to record the name as plain text is only the first of the choices that we have to make. Which writing system or alphabet are we going to

[2] For a thorough review of the multiple meanings of the word "text," please refer to (Sahle; Pierazzo).

use to record it? The Latin alphabet would be a common choice for people in Europe, but if we were to use that name in Japan it would be more natural to spell it using (comparable but not identical) Katakana characters. We will take the easy route and record this text using the Latin alphabet.

Character repertoire There are many ways to digitally encode a text written using the Latin alphabet in a document. The first thing to choose is a character repertoire, i.e. a standard that assigns a numeric code to each letter of the alphabet. For example, we can choose among ISO Latin-1 (ISO/IEC 8859-1:1998), Unicode (The Unicode Consortium), or CP-1252 (Microsoft Corporation). In this case we will choose Unicode and each letter will be represented by a so called Unicode *codepoint* a univocal numerical code, for instance, the letter b will be represented by the codepoint U+0062.

Character composition/decomposition In Unicode certain letters can be be encoded using various equivalent variants. In our case we have to decide how we want to encode the *ö* letter. Unicode gives us (at least) two possibilities: using the codepoint for ö (i.e., U+00F6) or using the combination of codepoints for the Latin letter o and the attached diaeresis (i.e., U+006F and U+0308). We will use the latter: separate codepoints for the letter and the diaeresis.

Byte encoding Now we must make yet another choice: which Unicode encoding should we use? In other words, how do we turn the numerical codepoints that Unicode associates with the letters into bytes? Unicode provides many possible encodings: UTF-8, UTF-16LE, UTF-16BE, UCS-32. In this example we will use UTF-8, an encoding that turns each codepoint into a group of bytes of variable length.

Byte endianness At this point, what is left to do is to turn the series of UTF-8 byte groups into a series of bytes and then into a series of bits. For this task, we will choose the so called little-endian order with 8-bit bytes. This series of bits (the so called *bitstream*) is what the computer will store on some permanent medium, for example on an hard drive.

Electron encoding However, bits are not physical entities *per se* and cannot be stored. In the case of an hard drive, bits must be stored as electric charges on a metallic plate; in the case of CDs, bits must be stored as opaque areas on the plastic substrate of the disc. In our case, we will use an hard drive whose chipset uses a simple kind of conversion from bitstream to electric states called 6b/8b (Wilamowski and Irwin). For example, the bits 110 will be stored as $-, +, +, +$.

Physical embodiment At this point no more choices are going to be taken. This series of electron states will be impressed by an electronic actuator on the platters of the disk. These semi-permanent alterations of the matter will be the physical carrier embodying our digital document.

Only after having gone through all these steps we can say that the business name “Böh & Son” has been serialized in an electronic document.

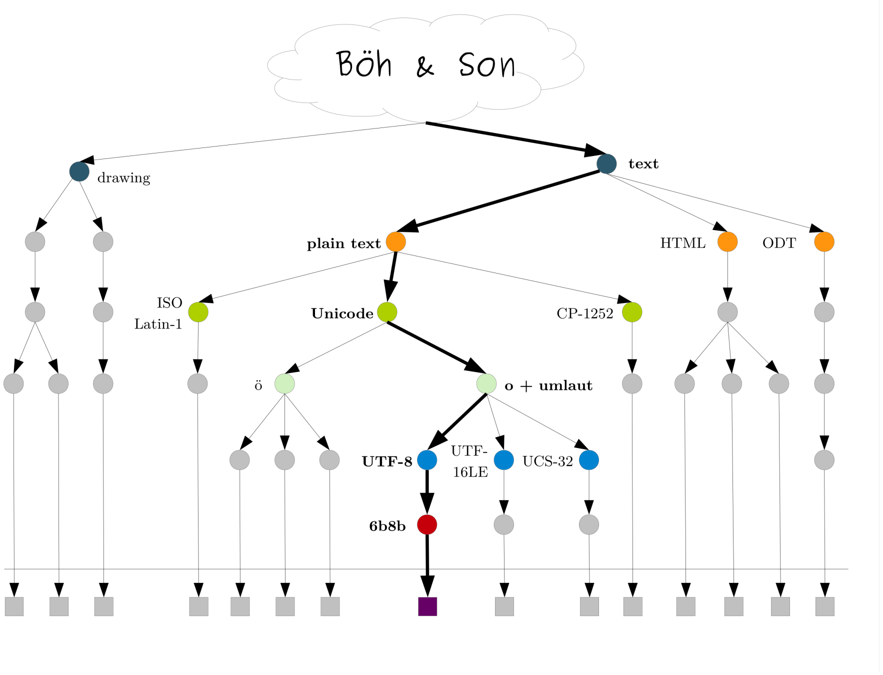


Figure 1: Tree of available choices during the creation and storage of a simple textual document. In bold the taken choices.

During the storage process we had to take many different choices. Figure 1 shows a tree of these possible choices, highlighting the choices that have been taken. In practice, however, most of these choices would not be taken by the users, but by a program, relying on clues from the user (e.g., "save the document as plain text") and following default choices hardwired in the source code by the developers (e.g., "use Unicode and UTF-8 when saving in plain text").

When an application will read this file, it will *deserialize* its content and basically undo the steps we made to write it. Pieces of information belonging to a less abstract level will be read, interpreted, and used to construct more abstract data structures, that will, in turn, be interpreted and used to construct even more abstract data structures. We see here a fundamental difference between serializing (writing) and deserializing (reading) a file. During the serialization of a document many choices are available and only few are taken. Instead, when a document is deserialized, only one set of choices can "explain" its set of physical signs (except few ambiguous cases).

We now proceed to see the details of the CMV+P model, using the example document we just created to illustrate it in practice.

3 The CMV+P model, linear version

This section describes the *linear version* of the CMV+P model, a reduced version used to describe documents whose content is not spread across different logical files.

The definitions of all the various concepts that comprise the CMV+P model will be given first. Afterwards, to illustrate how these concepts fit together in practice, the example document previously shown in section 2 will be reformulated using the CMV+P model.

3.1 Documents, abstraction levels and comparability

Definition (linear document)**.** A linear document $\mathfrak{D}$ is a potentially infinite stack of abstraction levels L_i:

$$\mathfrak{D} = (L_0, L_1, L_2, \dots)$$

For our purposes, we will limit ourselves to finite views on linear digital documents, so we will deal with documents of the form

$$(L_0, L_1, L_2, \dots, L_n)$$

where L_0 will always be the physical level and at least one of the abstraction levels will be a bitstream level.

The indexes 1, 2, …, n represent only the order in which levels are stacked in a certain document and are not meant to be compared among different stacks; in principle, the level L_3 in one document has nothing to do with the level L_3 in another document.

Our example document is thus represented by the following CMV+P document:

$$\begin{aligned} \mathfrak{D}_{ex} \quad = \quad & \Big(L_0^{physical}, L_1^{6b/8b}, L_2^{bitstream}, \\ & L_3^{UTF-8}, L_4^{Unicode}, L_5^{alphabet}, \\ & L_6^{plain-text}, L_7^{company-name} \Big) \end{aligned}$$

Definition (abstraction level). An abstraction level L is a tuple composed of a set of addressable elements C, a reference model M and a set of variants V:

$$L = (C, M, V)$$

Definition (content). The content of an abstraction level is a set C containing addressable elements and relations between them (e.g. order relations). The kind of elements that can be present in C and their structure are dictated by the model M.

Definition (model). The model of an abstraction level is a reference M to a specification that describes what are the types of the elements in C and what are the constraints of its structure.

Definition (variants). The set of variants of an abstraction level is a set V containing records of the choices, among those made available by the model M, made during the creation of C.

Definition (comparability). Two abstraction levels L_a and L_b are comparable if and only if they share the same model, i.e. $M_a = M_b$.

Definition (equality between levels). Two abstraction levels L_a and L_b are equal if and only if they are comparable and their contents are identical, i.e. $M_a = M_b$ and $C_a = C_b$.

Definition (equality between documents). Two documents $\mathfrak{D}_a$ and $\mathfrak{D}_b$ are equal if and only if only if they contain the same number of abstraction levels and all abstraction levels of the same index are equal, i.e. $\|\mathfrak{D}_a\| = \|\mathfrak{D}_b\| = n$ and $\forall i \in \{0, \ldots, n\}$ $\mathfrak{D}_a.L_i = \mathfrak{D}_b.L_i$.

Definition (equivalence between levels). Two abstraction levels L_a and L_b are equivalent under the equivalence relation eqv if and only if they are comparable and all the elements that are different in C_a and C_b have associated variants v_a, v_b and these variants are equivalent under eqv, i.e. $\exists\, (c_a, c_b) \in \delta\, (C_a, C_b) \leftrightarrow \exists v_a \in V_a, \exists v_b \in V_b, eqv\, (v_a, v_b)$.

3.2 An example document in CMV+P

We can now reformulate the “Böh & Son” document described in the previous section as a stack of CMV+P abstraction levels. The stack itself is depicted in figure 2.

Let us have a look more in depth at a couple of abstraction levels, starting with the alphabet abstraction level L_5^{alphabet}. The alphabetic abstraction level L_5^{alphabet} is composed of:

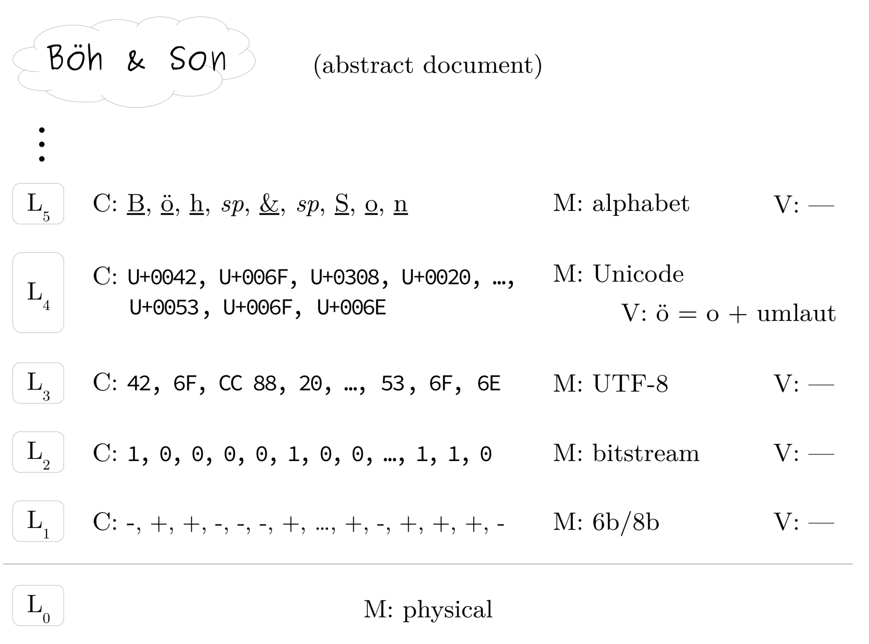

Böh & Son
(abstract document)
L5 C: B, ö, h, sp, &, sp, S, o, n M: alphabet V: —
L4 C: U+0042, U+006F, U+0308, U+0020, …, U+0053, U+006F, U+006E M: Unicode V: ö = o + umlaut
L3 C: 42, 6F, CC 88, 20, …, 53, 6F, 6E M: UTF-8 V: —
L2 C: 1, 0, 0, 0, 0, 1, 0, 0, …, 1, 1, 0 M: bitstream V: —
L1 C: -, +, +, -, -, -, +, …, +, -, +, +, +, - M: 6b/8b V: —
L0 M: physical

Figure 2: Abstraction levels for the "Böh & Son" plain-text document described in section 2.

- C_5, that contains an ordered list of letters (more precisely, *graphemes*), chosen among those defined in the Latin alphabet;
- M_5, a reference to the rules of the Latin alphabet and writing system (i.e. documents are composed of certain letters and punctuation signs arranged in a certain order);
- V_5, an empty set (the Latin alphabet model does not provide different but equivalent variants among which one can choose, therefore there are no choices to be made at this level of abstraction).

More formally, L_5^{alphabet} can be represented as

$$\begin{aligned} L_5^{\text{alphabet}} &= (C_5, M_5, V_5) \\ C_5 &= (\underline{\text{B}}, \underline{\text{ö}}, \underline{\text{h}}, sp, \underline{\&}, sp, \underline{\text{S}}, \underline{\text{o}}, \underline{\text{n}}) \,, \\ M_5 &= \text{Latin alphabet}, \\ V_5 &= \{\} \end{aligned}$$

The second level we will look at is the Unicode abstraction level L_4^{Unicode}. At the Unicode level, the content is a series a so-called *codepoints*, numerical identifiers for specific glyphs (i.e. letters) as specified in the Unicode repertoire. The Unicode repertoire is a compilation of letters from many different writing systems. For instance, the codepoint allocated for the Latin letter capital A is U+0041, the codepoint for the Greek letter small beta (i.e. β) is U+03B2. For certain letters, Unicode allows for more than one codepoint, or combinations of codepoints. The Latin letter small O with diaeresis (i.e. ö) is one of these cases: it can be encoded using the single codepoint U+00D6 or the combination of codepoints U+006F and U+0308, respectively Latin letter small O and the combining diaeresis. In our example we decided to use the combining form. This will be reflected in C_4 and V_4: in C_4 two codepoints will be used to encode the letter ö; in V_4 we will record this choice.

$$
\begin{aligned}
L_4^{\text{Unicode}} &= (C_4, M_4, V_4) \\
C_4 &= \big(\text{Unicode-codepoint(U+0042)}, \\
&\quad \textbf{Unicode-codepoint}(\mathbf{U} + \mathbf{006F}), \\
&\quad \textbf{Unicode-codepoint}(\mathbf{U} + \mathbf{0308}), \\
&\quad \text{Unicode-codepoint(U+0020)}, \\
&\quad \ldots \\
&\quad \text{Unicode-codepoint(U+006F)}\big), \\
M_4 &= \text{Unicode, version 7.0}, \\
V_4 &= \big\{\big(\text{encode } \underline{\text{ö}} \text{ as o}^U \text{ + combining diaeresis}^U\big)\big\}
\end{aligned}
$$

3.3 Transformation functions: encoding and decoding functions

When a document is read, saved or edited, the content of all the abstraction levels that comprise a document must be kept in sync. It is thus necessary to have mechanisms that can move the content across different abstraction levels, from the document as seen via the interface by the user to the document as stored in the physical medium and vice versa. In CMV+P this mechanism is fulfilled by transformation functions.

Transformation functions are used to transform content stored according to the model of a certain abstraction level into content stored according to the model of another abstraction level. Transformation functions used during the serialization phase are called *encoding functions*, those used during the deserialization phase are called *decoding functions*. During the serialization phase, encoding functions are used to turn the content of a more abstract level into content suitable for the next less abstract level. The very last encoding function is responsible for implanting the document into the physical carrier. During the deserialization phase, conversely, decoding functions are used to turn the serialized content into abstract data structures that the applications can work with.

Definition (transformation function)**.** A transformation function $trans$ is a function that transforms the content C_a of an abstraction level L_a (created according to model M_a and variants V_a) into the content C_b of an abstraction level L_b, created according to the model M_b and the variants V_b.

$$trans : (C_a, M_a, V_a, M_b, V_b) \to C_b$$

While the term *function* is used, it must be noted that not all transformation functions are bijective functions (i.e. complete and reversible). Some transformation functions related to the most abstract levels may not even be proper functions in a strict mathematical sense. The impact of various properties of the transformation function (e.g. bijectivity, calculability, reversibility) on the creation and interpretation of the document is out of scope for this introductory article and will be discussed in a future publication.

Transformation functions in practice

In concrete applications, the role of the transformation functions is fulfilled by various pieces of code, often embedded in shared libraries. The complexity of these function ranges from trivial to extremely intricate. For example, the encoding function from L_4^{Unicode} to $L_3^{\text{UTF-8}}$ can be written in a handful of lines of code, while the encoding function from L_5^{alphabet} to L_4^{Unicode} could consist of thousands of lines spanning a dozen libraries. A consequence of this is that, given any two abstraction levels, there exist many different concrete encoding and decoding functions between them and, in theory, an infinite number of transformation functions is possible.

Another difference between the theory and the reality is that, in theory, encoding functions and decoding functions are the mathematical inverse of each other while, in practice, the implementation of the encoding function may bear no resemblance to the implementation of the specular decoding function. Take for example a hypothetical plain-text editor software. It displays the letters that make up the text, so it must deal with L_5^{alphabet}. At the same time, the editor internally processes the textual data as Unicode codepoints at abstraction level L_4^{Unicode}. It follows that the editor must have a pair of encoding/decoding functions for these levels: an encoding function that serializes the letters of L_5^{alphabet} into the codepoints of L_4^{Unicode}, as well as a specular decoding function to deserialize L_4^{Unicode} into L_5^{alphabet}. In concrete terms, in this editor the encoding function is the code that turns input signals from the operating system (in forms of keystrokes) into a equivalent data structures that hold sequences of Unicode codepoints; the decoding function, instead, is the code that turns the data structures that hold the Unicode codepoints into the data structures that describe the letters (or graphemes) to be displayed on the screen using an appropriate font. It is clear that these two pieces of software have little in common.

4 Using CMV+P to compare documents

Now that we have seen the basics of the model, we can move on to show how CMV+P helps in comparing documents in practice. Comparison algorithms and tools can use CMV+P to

- identify at which abstraction levels it is possible to compare two documents;
- classify which parts are identical, equivalent or different at one or more abstraction levels;
- understand which measures should be taken to compare two ostensibly incomparable documents.

To illustrate these points, this section presents a few examples of increasing complexity.

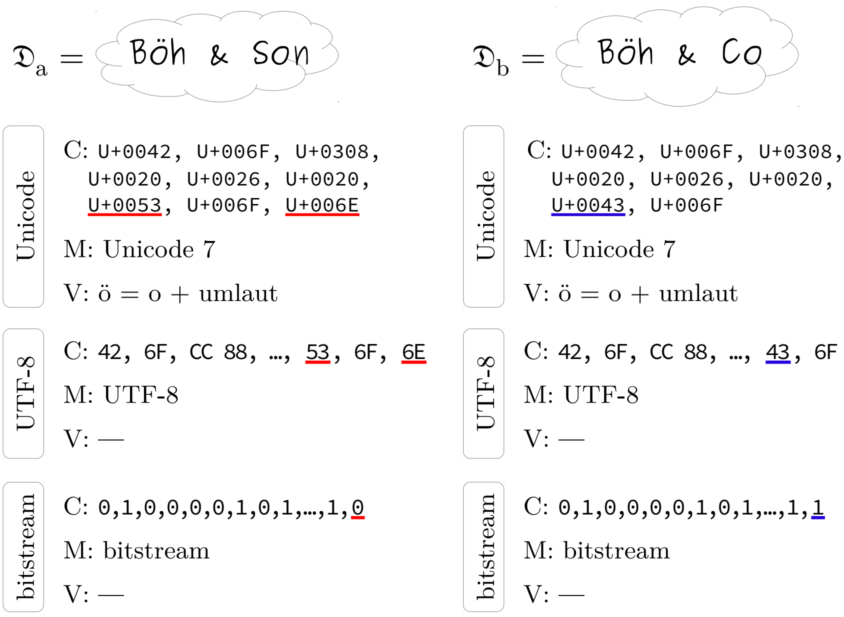
In the first two examples, the plain-text document discussed in the previous sections is compared with two slightly modified copies. Here, various kinds of differences in content and variants are analyzed. The third example shows a comparison between the same ODT file and an HTML file with similar content. This last example delves into the idea of comparing documents in different formats.

The fourth and last example deals with comparing "incomparable" documents and the associated paradox of the "equal but different" files, discussed at the beginning of this article. The purpose of this last example is to demonstrate that tools that use CMV+P can leverage their knowledge of the stacks of abstraction levels to make documents comparable by, for example, passing them through extra transformation functions.

These examples show only a few of the practical applications of the CMV+P model. Additional, more complex practical aspects of the model will be explored in future publications.

4.1 Identification of differences

Our first example deals with an elementary case of difference: a textual substitution. The first document $\mathfrak{D}_a$ contains the text "Böh & Son," the second document $\mathfrak{D}_b$ contains the text "Böh & Co." Both files are plain-text documents and have been encoded using Unicode and UTF-8. Figure 3 shows the CMV+P stacks for these two documents.



$\mathfrak{D}_a$ = Böh & Son
Unicode
C: U+0042, U+006F, U+0308, U+0020, U+0026, U+0020, U+0053, U+006F, U+006E
M: Unicode 7
V: ö = o + umlaut
UTF-8
C: 42, 6F, CC 88, …, 53, 6F, 6E
M: UTF-8
V: —
bitstream
C: 0,1,0,0,0,0,1,0,1,…,1,0
M: bitstream
V: —
$\mathfrak{D}_b$ = Böh & Co
Unicode
C: U+0042, U+006F, U+0308, U+0020, U+0026, U+0020, U+0043, U+006F
M: Unicode 7
V: ö = o + umlaut
UTF-8
C: 42, 6F, CC 88, …, 43, 6F
M: UTF-8
V: —
bitstream
C: 0,1,0,0,0,0,1,0,1,…,1,1
M: bitstream
V: —

Figure 3: CMV+P stacks for two plain-text files with slightly different content.

A non CMV+P-based diff tool can focus on only one of the three abstraction levels shown in figure 3. For instance, a binary diff tool will compare only the bitstreams, while a classical text comparator will focus only the sequence of Unicode codepoints.

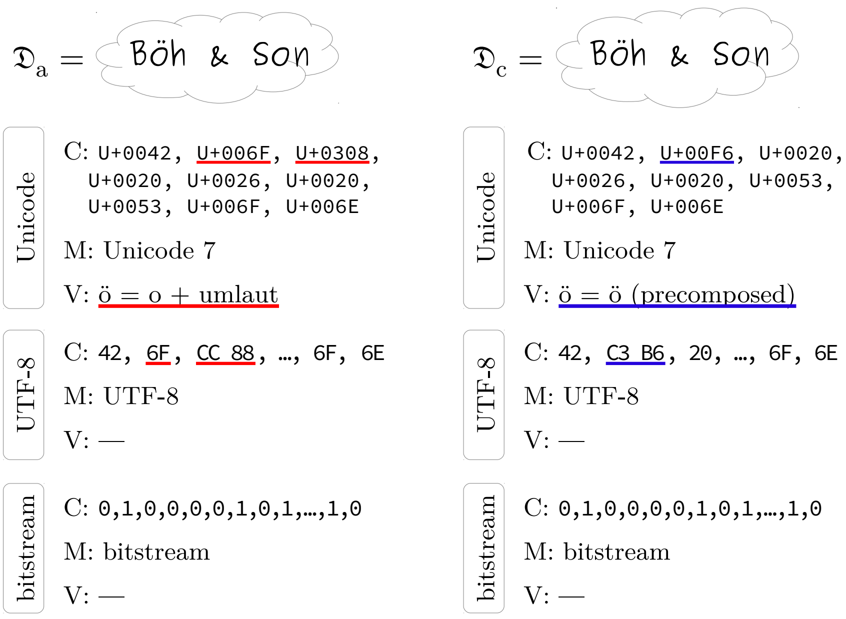
CMV+P allows tools to have a more holistic view of the differences. CMV+P-aware tools can provide a different set of differences for each abstraction level. For example, a tool could say: “There are three different sets of differences: at the bitstream level these bits have been changed; at the UTF-8 level these groups of bytes have been changed; at the Unicode level these codepoints have been changed.” With the appropriate user interface, a single tool could provide the users with the exact kind of information they are after: the author of the text may be interested in seeing which words have changed, whereas the developer of a text-editor that is debugging a UTF-8 problem may be interested in seeing the changes expressed in terms of UTF-8 groups.

In more complex file formats, a CMV+P-aware tool has the ability to show changes at many more levels, in a clear and unambiguous way. For example, when comparing HTML files, a tool could show differences in their rendering, differences between their XML trees, differences in the textual content of various elements, or differences between XML serializations, just to name a few.

4.2 Equality, equivalence and difference

The second example illustrates how the concepts of equivalence and equality can be precisely expressed and managed thanks to the variants set V recorded in each CMV+P abstraction level.

The documents compared in this example are the plain-text document $\mathfrak{D}_a$ of the previous example and a copy of it, $\mathfrak{D}_c$, that has been serialized using the single precomposed Unicode character *ö* instead of the sequence *o + combining diaeresis*. Figure 4 shows the CMV+P stacks for $\mathfrak{D}_a$ and $\mathfrak{D}_c$.



$\mathfrak{D}_a$ = Böh & Son
Unicode
C: U+0042, U+006F, U+0308, U+0020, U+0026, U+0020, U+0053, U+006F, U+006E
M: Unicode 7
V: ö = o + umlaut
UTF-8
C: 42, 6F, CC 88, …, 6F, 6E
M: UTF-8
V: —
bitstream
C: 0,1,0,0,0,0,1,0,1,…,1,0
M: bitstream
V: —
$\mathfrak{D}_c$ = Böh & Son
Unicode
C: U+0042, U+00F6, U+0020, U+0026, U+0020, U+0053, U+006F, U+006E
M: Unicode 7
V: ö = ö (precomposed)
UTF-8
C: 42, C3 B6, 20, …, 6F, 6E
M: UTF-8
V: —
bitstream
C: 0,1,0,0,0,0,1,0,1,…,1,0
M: bitstream
V: —

Figure 4: CMV+P stacks for two plain-text files. In $\mathfrak{D}_a$, ö is encoded with a Unicode combining character, in $\mathfrak{D}_c$ with a precomposed character.

A non CMV+P-based diff tool can either say that $\mathfrak{D}_a$ and $\mathfrak{D}_c$ are different (e.g., if it compares the bitstream of the two documents) or equal (e.g., if it prenormalizes the documents with one of the procedures suggested by the Unicode consortium (Davis and Whistler). Which of these two answers is correct depends on the needs of the user.

A CMV+P-based diff tool can, instead, provide a more complete view of the results of the comparison. It can state without ambiguity that:

- the alphabetical levels of $\mathfrak{D}_a$ and $\mathfrak{D}_b$ are identical,
- their Unicode levels are different but equivalent, and
- both their UTF-8 levels and their bitstream levels are different.

The fact that CMV+P keeps track of the set of variants used in the serialization of the documents allows formal and unambiguous definitions of what is identical and what is equivalent; c.f. the definitions in section 3. In turn, the availability of these definitions simplifies and streamlines the creation of diff tools where most of the code is format- and model-agnostic. The model-specific parts are confined to small function eqv that check the equivalence between elements that have an associated variant in the V set. Usually these functions are provided in the specifications of the model.

Performance improvements are also made possible by the existence of the variants set. Only the few variants in V need to be checked using expensive equivalence checks, the rest of the elements in C can be tested with fast equality checks.

4.3 Comparison between different formats

Allowing documents in different formats to be compared is another of the strengths of the CMV+P format. Normally, documents stored in different formats cannot be compared. For example, an HTML file cannot be compared with an ODT file, although both are basically text files with the possibility of embedding images. This example demonstrates how the use of CMV+P allows a diff tool to reason over the structure of the files being compared and to find abstraction levels that can be compared.

For this example, we will need more complex documents than the plain-text files used in the previous sections. The first document in this example is a file produced using LibreOffice in the so-called "compressed flat OpenDocument Format" (commonly referred to as "compressed flat ODT"; in the rest of this example just "ODT"). In this document, there is only a heading with the name of the business we already used in section 2: "Böh & Son." The second document is an HTML5 document with the same textual content. The CMV+P stacks of these two documents are depicted in figure 5.

Here we see that the abstraction levels of the two documents can be classified in three ways:

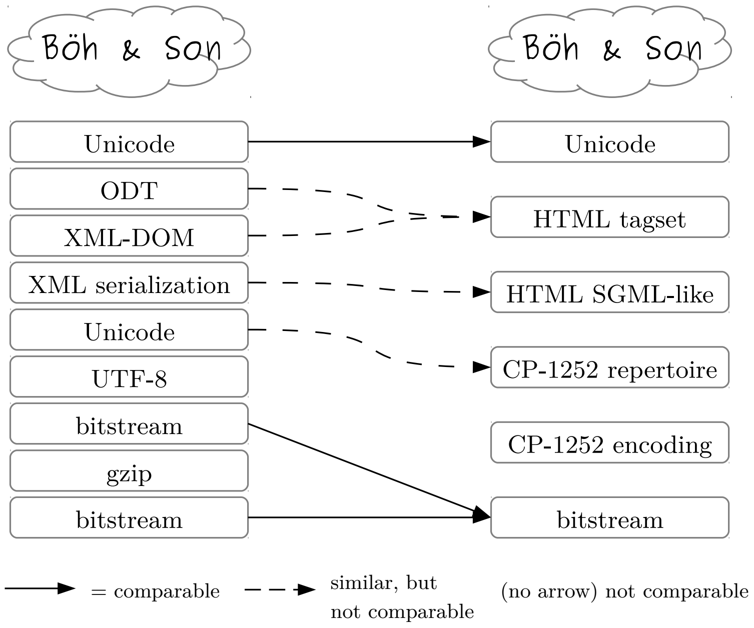


Figure 5: CMV+P stacks for an ODT and an HTML document. Only a few levels can be compared.

1. **Comparable** levels that can be compared directly because they share the same model. For instance, the Unicode levels or the bitstream levels.
2. **Similar, but not comparable** levels whose content is somehow related but that has been encoded using different models. For example, both the ODT structures and the HTML tagset have the concepts of a "heading" or a "paragraph," although they are expressed in different ways. An advanced comparison tool could compare across these similar abstraction levels if it did know about both models and had some kind of conversion function that could be used to remodel the content of these levels.
3. **Incomparable** levels whose models deal with completely different concepts, for instance gzip and HTML.

Thanks to the CMV+P model, it becomes clear at which levels comparisons can be done, where conversion functions could make two levels comparable and for which levels no comparison is possible at all.

4.4 The “equal but different” paradox solved with CMV+P

CMV+P solves the paradox enunciated in the introduction: a file and a compressed copy of it are completely different, even though they contain exactly the same content.

Let’s take the plain-text document introduced in the first example (section 4.1) and compress a copy of it with gzip. The CMV+P stack of these two documents are shown in figure 6.

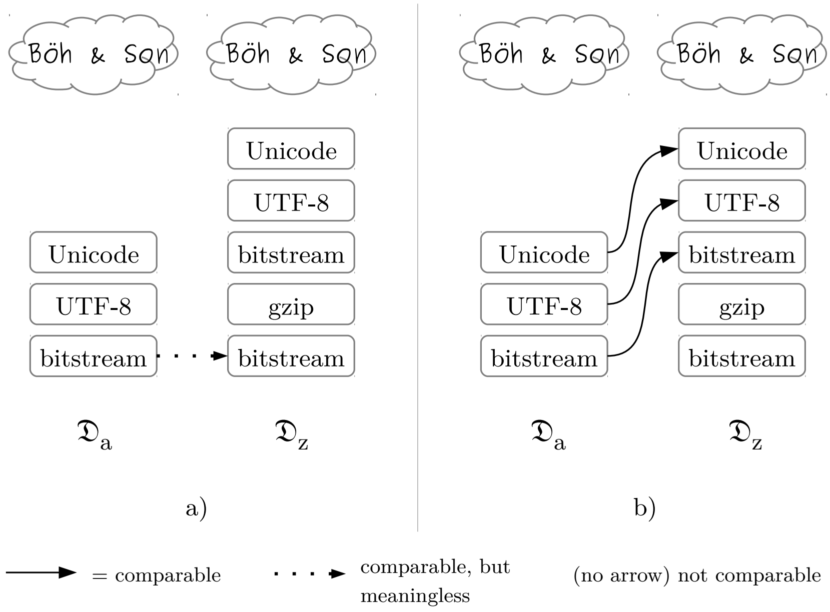


Figure 6: CMV+P stacks of $\mathfrak{D}_a$ and $\mathfrak{D}_z$. $\mathfrak{D}_a$ is a plain-text file; $\mathfrak{D}_z$ is a copy of $\mathfrak{D}_a$ that has been compressed with gzip. Subfigure a) shows how a CMV+P-aware diff tool would compare the two documents; subfigure b) shows the alignment between abstraction levels found by a CMV+P-aware tool.

All a non CMV+P-aware diff tool can do is compare the bitstream levels of $\mathfrak{D}_a$ and $\mathfrak{D}_z$. These two bitstream levels are indeed comparable, but the result of their comparison is meaningless: the series of serialized characters of $\mathfrak{D}_a$ is being compared to the quasi-random sequence of bits that is the compressed file $\mathfrak{D}_z$.

In contrast, a CMV+P-aware diff tool notices that there is a mis-alignment between the two CMV+P stacks and understands that many different meaningful comparisons are possible if the proper alignment is restored. In this particular case, the diff tool should start the comparison process after the bitstream of $\mathfrak{D}_z$ is decompressed (or, in more precise terms, after the gzip level of $\mathfrak{D}_z$ has been deserialized using a gzip-to-uncompressed-bitstream decoding function).

A side note: aptly, in order to understand which operations are needed to make two documents comparable, CMV+P-aware diff tools (Barabucci, "diffi") must perform a sequence-alignment between the two stacks, the exact task that is at the base of almost every comparison algorithm. In other words, they have to "diff the stacks."

5 Conclusions

This paper introduced the CMV+P model (linear version) and showed that digital documents exist simultaneously at different abstraction levels.

Each abstraction level has its own peculiarities but all abstraction levels can be formally described in terms of Content, Model and Variants, together with the associated encoding and decoding functions. The final P in CMV+P reminds us that digital documents are also Physical documents, although their nature requires the use of software mediators to manipulate them.

The CMV+P model is especially useful in the context of document comparisons, in particular comparisons done with computer tools. The CMV+P model allows humans and computer tools to identify with precision

- at which abstraction levels of an electronic document a change has been detected,
- which elements of these abstraction levels have to do with this change,
- and, in general, which comparisons are possible between two electronic documents.

Bibliography

Barabucci, Gioele. "Introduction to the universal delta model." *ACM Symposium on Document Engineering DocEng '13, Florence, Italy, September 10-13, 2013*, edited by Simone Marinai and Kim Marriott, 2013, pp. 47–56, doi.acm.org/10.1145/2494266.2494284. Accessed 20. Feb. 2018.

———. "diffi: diff improved; a preview." *Proceedings of the ACM Symposium on Document Engineering 2018, DocEng 2018, Halifax, NS, Canada, August 28-31, 2018*, 2018, pp. 38:1–38:4, doi.org/10.1145/3209280.3229084. Accessed 20. Sept. 2019.

Bray, Tim et al. "Extensible Markup Language (XML) 1.0 (Fifth Edition)." *Recommendation, W3C*, November 2008, www.w3.org/TR/2008/REC-xml-20081126/. Accessed 20. Feb. 2018.

Davis, Mark and Ken Whistler. "Unicode normalization forms. Standard Annex 15." *Unicode*, May 2017. unicode.org/reports/tr15/. Accessed 20. Feb. 2018.

Freed, N. and N. Borenstein. "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types." *RFC 2046 (Draft Standard)*, November 1996, www.rfc-editor.org/rfc/rfc2046.txt.

Accessed 20. Feb. 2018. Updated by RFCs 2646, 3798, 5147, 6657, 8098.

Gailly, Jean-loup and Mark Adler. "GNU Gzip." *GNU Operating System*, 1992. www.gnu.org/software/gzip/. Accessed 20. Feb. 2018.

"ISO 26300-1:2015. Information technology – Open Document Format for Office Applications (OpenDocument) v1.2 – Part 1: OpenDocument Schema." *Standard, International Organization for Standardization*, 2015.

"ISO 32000-1:2008. Document management – Portable document format – Part 1: PDF 1.7." *Standard, International Organization for Standardization*, 2008.

"ISO/IEC 8859-1:1998. Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1." *Standard, International Organization for Standardization*, April 1998.

Microsoft Corporation. "Code page 1252 windows latin 1. Technical report, ANSI." *Microsoft Developer Network*, 1998. msdn.microsoft.com/en-us/library/cc195054.aspx. Accessed 20. Feb. 2018.

Pierazzo, Elena. *Digital Scholarly Editing: Theories, Models and Methods*. Routledge, 2015.

Sahle, Patrick. *Digitale Editionsformen: Textbegriffe und Recodierung*. Schriften des Instituts für Dokumentologie und Editorik, vol. 9 Instituts für Dokumentologie und Editorik, Books on Demand, 2013.

The Unicode Consortium. "The Unicode Standard. Technical Report Version 6.0.0." *Unicode Consortium*, 2011. www.unicode.org/versions/Unicode6.0.0/. Accessed 20. Feb. 2018.

Wilamowski, Bogdan M. and J. David Irwin. *Industrial Communication Systems*. CRC Press, Inc., 2nd edition, 2011.