PDF/A-1a in ConTEXt-`mkiv`

Luigi Scarso

Abstract

I present some considerations on electronic document archiving and how ConTEXt-`mkiv` supports the ISO Standard 19500-1 Level A Conformance (PDF/A-1a:2005), an ISO standard for long-term document archiving.

Abstract

Alcune considerazioni sulle problematiche dell'archiviazione di documenti elettronici e come ConTEXt-`mkiv` supporta lo standard ISO 19500-1 Level A (PDF/A-1a:2005) relativo all'archiviazione digitale documentale.

1 Introduction

In this paper I will try to illustrate some aspects of electronic documents archiving starting from the position that it's fundamentally a typographic language problem, and also, in the contest of information technology, a programming language one. After some theoretical considerations, I will show some important typographic languages that are used, and then I will briefly talk about the ISO Standard PDF/A-1 and how ConTEXt-`mkiv` tries to adhere to its requirements. About the typographic style of this paper, I will follow these simple rules: I will avoid footnotes and citations on running text, and I will try to limit lists (e.g. only itemize and enumerate) and figures; the last section before the References one will collect all citations.

2 Some theoretical considerations

The main problem of electronic document archiving is to find a good answer to the question: "Will we be able to read it again in the next x years or, better, forever?" There are basically two kinds of requests: just to be able to understand the document contents, or to be able to fully and faithfully reproduce the original document. With electronic documents we aim at the second one.

It should be clear that it is not a problem to make an identical copy of an electronic document, but to obtain a reasonable confidence that we will be able to read the present document as originally intended. So the problem is to define an electronic format which is clear enough to allow a correct implementation of a consumer program both now and in the future, hence robust against accidental errors and independent of external resources, i.e. *self contained*, and semantically adequate for the purpose of document archiving.

The first requirement calls for a widely accepted standard; the semantically adequacy calls for a (formal) language, and precisely a *Typographic Language*. Some authors also distinguish between digital typography (the design and rendering of a "character"), micro typography (which covers aspects of type and spacing, such as kerning between letters, ligatures, line breaking etc.), and macro typography (that covers the visual quality of the document, hence the design of headings, lists, pages but also colors and images). A formal typographic language ideally covers all these aspects while being also a programming language.

We actually need to preserve both the content and the visual appearance of the document and inevitably this demands for a language that is able to talk of fonts, colors, images and *animations* in an abstract fashion and easily connects these abstractions to the concrete world of printing (and not only printing on paper), i.e. it must be a *Page Description Language*.

But this is not enough: we need another more abstract connection to the world of semantic and structure. As of today, we are still unable to exhibit a practical algorithm that checks the semantic correctness of a document in a human language, and there are some problems with the syntax: we can only hope to check the semantic of the typographical unit, the "character". This seems more affordable: just a (possibly unlimited) list of `(glyph, id, semantic)`, where `glyph` is the pictorial representation of a "character" and `id` is a unique identifier to prevent misunderstanding (for example a space is also a "character" and there are several kinds of spaces that we need to distinguish). The typographic language hence can use the `id` both to display the "character" (its `glyph`) and as a reference to its `semantic`. But here we enter into a cultural and linguistic domain: for example the `semantic` require a standardized metalanguage and more often than not a glyph is intimately tied with the literature of a given cultural area (of the present as of the past): it's not unusual to see the same glyph in completely different contexts that must be clearly distinguished. It's hence reasonable to expect practical/contingent conventions hard to implement or to respect — and perhaps also meaningless in the future. Even the concept of lists may be impractical: some glyphs are combinations of others "characters" (basic signs and/or other glyphs) and the rules of combinations can lead to infinite possibilities: that means that we need a standard *Character Language*.

The structure of the document calls for a *Markup Language*. This field is better known and well established: the key points are a clear separation between structure and contents, and the usage of a standard metalanguage. This is a peculiar property of an electronic document: the structure in a paper document can be inferred from the physical copy, but it's not embedded in it. Markup languages permit computational classification of electronic documents, high degree of content reuse (e.g. automatic speaking), efficiency in storage, hyper link capabilities, while preserving the original document unaltered — things that are difficult or impossible to achieve in traditional documents.

The controversial point is about the semantic: given a markup language, what is its domain? Is it wide enough to cover all the aspects of our document? Is it infinite or finite? For example, a markup language about the book's structure can cover only a part of all the aspects of a technical drawing or an invoice document. We can think of a sort of a universal infinite markup language that covers all the aspects, but sentences of an infinite language can be long enough to become a practical problem, and, most important, markup languages are hard to design. Ideally, we would like a language defined by a compact grammar, with infinite sentences and a wide semantic; practically we must rely on the human common sense of measure and adaptability.

So far we have seen that a typographic language alone is not sufficient to honor the contents. We need other kinds of languages. But even with them we need something else to be able to reproduce the document as originally intended: we must be reasonably certain that the current document *is* the original document. Traditional paper documents are always tied to a physical support: we have only one original and one or more copies. Sometimes the copies are so faithful that the original seems unimportant (think of a newspaper) and sometimes things can be arranged to produce two or more "originals" — but it is another way to say that there is one original and one o more copies that are "indistinguishable from the original for practical purpose".

Things change radically with the electronic documents: the copy is indistinguishable from the original, and it's easy to verify if two documents are equal — just a comparison between bytes — as to keep track of changes (the history of the document). But the negative side is that it's now more difficult to decide the right one between two documents when both claim to be the "original one".

Again, it's another peculiar property of any electronic document that solves the problem: the property to be seen as a *number*. Each stream of bits can be easily analyzed and transformed by means of a particular kind of mathematical functions, among which the *one-way function* plays a central role: it is a function that is *easy* ("cheap enough for the legitimate users") to calculate for every input but *hard* ("prohibitively expensive for any malicious agents") to invert given the image of a random input. A trivial example: it is easier to multiply two large numbers than to find the divisors of a large number.

A one-way function can be used to encode a stream of bits so that only who knows the key can decode it, but it can also be used to detect and prevent the modifications of the stream itself — and this is what we need for the document archiving. It's the modern edition of the signet ring, in facts it is called *digital signature*, but with a subtle difference: up today a formal proof does not exist that inverting an one-way function will *always* be a hard task. Unfortunately the history just says the opposite: with the raising of computational power and the advancement of modern algebra some of the early one-ways-functions were "easily" (for a malicious agent) inverted so that literally every day the security experts must check the news. With Internet there are more potential malicious agents today than in the past, software and documentation are available for free, more computational power can be achieved with clustering (often abusively). And, finally, for a casual user managing a secret key can be problematic.

It's important to note that the digital signature depends only on the digital nature of the content, and not from any typographic property: every stream of bits (i.e. every kind of file) can be encrypted.

3 Some typographic languages

In the previous section we delineated some properties of a typographic language for document archiving: now let's see some real candidates.

3.1 SVG and XSL-FO

The Scalable Vector Language is a W3C recommendation for describing two-dimensional graphics both static and dynamic. It's a vector graphics markup language in XML format and hence it can express concepts like fonts, colors, curves but it's not a strictly page description language — every SVG graphics has exactly one page, even if multiple layers are possible. A W3C draft extending the standard SVG with the notion of pages has been written, but development efforts are now directed to the next release of SVG, so it's unlikely that this draft will have further influences in the present recommendation. Moreover, there are no ways to express the logical structure of a page.

Another W3C recommendation that is related to typography is the Extensible Stylesheet Language, an XML application that defines a language for transforming XML documents (XSLT) and an XML vocabulary to specify formatting semantics, informally referred to as XSL formatting objects or XSL-FO.
The semantic of XSL-FO is mostly related to the layout of a document; there are some structural elements especially for book-like documents (i.e. tables, lists, but not sectioning), and hence it looks like a natural companion of SVG because, thanks to the namespace, we can compose documents with fragments from different markup languages, while preserving the syntactic and semantic correctness of each language.

SVG and XSL have some good points: they are standards according to a widely recognized world organization, they are free from royalties and freely available, and there are software tools to check the syntactic correctness. At least for SVG, the Inkscape program is a quite good editor for an average use and the last release has also the interesting feature of embedding the JavaScript language into a graphic, so that it's possible to consider SVG + JavaScript as a full typographic language. As for any XML application, they both use the Unicode standard for the character encoding.

But as of today their diffusion is still limited: one of the most important browser, Internet Explorer 8, still lacks support for SVG (it should be supported in the next version 9) and XSL-FO is quite simple in its typographic capabilities to gain popularity *per se* even if we consider that XSLT is a programming language — but intended to transform generic XML documents, not specific for typography. For the document archiving purpose a document markup language with a richer semantic like DocBook should be preferable, but managing three different standardized languages is not a good solution, given that each one evolves independently; on the other side just two of them are not enough.

3.2 TEX

TEX is a *typographic macro-programming Turing-complete language* by D. Knuth. The original language is described by the author in The TEXbook, and it consists of about 300 primitive commands that cover, in essence, how to organize the characters in rows, the rows in lines and the lines in pages, the indexing and the tables. In the same book it is also described the *plain format*, a collection of almost 600 macros built on top of the primitive commands, that provides sectioning and a few other typographic constructs (and used to typeset the book itself). It has only a basic notion of graphics, but the macro `\special` permits to implement extensions: particularly important is the extension to the PostScript language, a page description language which is also a full-featured programming language.
The key point is the Turing-completeness. With TEX we can build an arbitrary format that is, basically, a document markup *and programming* language and this definitely solves the problem to choose the right markup language for document structure because new structures can be build remaining inside the format. The macro nature of the language is also well suited to process the input, so that it's possible to build macros that manage a particular text encoding — theoretically all computable *character languages*. Knuth also designed a compact page description language, the *DeVice Independent* format, or DVI, that represents the output of a `tex` file as processed by the TEX program, and the METAFONT programming language to design fonts (i.e. it's a *digital typographic programming language*), which is the companion program to produce bitmap fonts. Finally he also described the complete implementation in PascalWeb of all these programs. TEX was so accurately designed and so deeply tested on a wide range of hardware/software (still today) that we can consider it as practically bug free.

On the other side, TEX was not designed for archiving purpose. The DVI format is not self contained (for example, the fonts aren't included) and this is true even if we consider the TEX source file as an electronic format. There is no standard organization behind TEX and the original program today is surpassed by new implementations: the most important are pdfTEX, XƎTEX and luaTEX. Finally, TEX is almost unknown outside the scientific community, even if its hyphenation algorithm is widely used.

3.3 PDF

The Adobe Portable Document Format (PDF) is the successor of the PostScript language, a well known and established *page description language* for printing documents. Basically it extends the PostScript model by adding interactive features as navigation and annotations (these are quite similar to a static `html` page with a script language similar to php or JavaScript; in fact PDF uses JavaScript), tree dimensional images, movies and animations (all for screen documents), a complete support for Unicode, a new font technology, digital signature, a document meta-markup language and a simple html-like markup language and a radically different electronic format — a binary format instead of textual.

Adobe started PDF around 1993 and until now, following the same line of conduct of PostScript all specifications are publicly available, as there is a pdf reader free for downloading (the full featured pdf editor Acrobat is available as a commercial product). In 2008 the PDF version 1.7 became the ISO standard 32000-1:2008, and as of today Adobe has promulgated two extensions (for Rich Media contents and forms) which probably will be included into the next ISO standard 32000-2 under development.

There are many good points in PDF. Practically all electronic documents can be converted in a self contained PDF. Thanks to PDF binary format, exchanging and archiving are more reliable because even the modification of only one byte may entail an error when reading. Editing is also difficult (but nowadays less than in the past) and so accidental modifications; conversely producing PDF is increasingly simple and there are now better pdf readers other than the Adobe one (but currently very few commercial products can compete with the Adobe pdf editor). The print quality of PDF is the same of PostScript but a pdf file can also embed the logical structure and finally, most important, pdf documents are enormously widespread all around the world, fitting well with the local typographic traditions. It's a winning ISO Standard.

But there are also some limitations. PDF is not a programming language: unlike PostScript, for example, it needs another language (the *Job Description Format*) to describe a print job, and unlike TEX we cannot use it as a typographic language. In the end, it's not so different from SVG: even Adobe has started the *Mars* project to give "an XML-friendly representation for PDF documents called PDFXML", and it's not necessary to invent a sort of binary SVG because the digital signature can be used to detect and prevent document modifications. Finally it seems that by the end of this year all the most important browsers will support the SVG format to some extent, so they can be insidious competitors for the pdf reader (not for printing, anyway).

4 The PDF/A-1 ISO Standard

Probably one of most known PDF version is `PDF 1.4` (around 2001, almost ten years ago) maybe because the companion Acrobat 5.0 was a robust programs and the pdf Reader was freely available for several platforms both as a program and as plugin for browsers. We keep having a huge amount of electronic documents that are in `PDF 1.4`, hence we should not be surprised if Adobe pushed it as reference for document archiving. What follows is a verbatim copy from http://www.digitalpreservation.gov/formats/fdd/fdd000125.shtml and it's a good description:

PDF/A-1 is a constrained form of Adobe PDF version 1.4 intended to be suitable for long-term preservation of page-oriented documents for which PDF is already being used in practice. The ISO standard [ISO 19005-1:2005] was developed by a working group with representatives from government, industry, and academia and active support from Adobe Systems Incorporated. Part 2 of ISO 19005 (as of September 2010, an ISO Draft International Standard) extends the capabilities of Part 1. It is based on PDF version 1.7 (as defined in ISO 32000-1) rather than PDF version 1.4 (which is used as the basis of ISO 19005-1).

PDF/A attempts to maximize device independence, self-containment, self-documentation. The constraints include: audio and video content are forbidden, JavaScript and executable file launches are prohibited, All fonts must be embedded and also must be legally embeddable for unlimited, universal rendering, colorspaces specified in a device-independent manner, encryption is disallowed, use of standards-based metadata is mandated.

The PDF/A-1 standard defines two levels of conformance: conformance level A satisfies all requirements in the specification; level B is a lower level of conformance, "encompassing the requirements of this part of ISO 19005 regarding the visual appearance of electronic documents, but not their structural or semantic properties".

In essence the standard wants to ensure that every typographic element, from the low level character to the high level logical structure is unambiguously defined and unchangeable — and it does, it achieves its purpose: every character must be identified by a Unicode id, which is an international standard and a *character language*, every color must be device independent by means of a color profile or output intent, there must be precise metadata informations for classifications and the document must have a logical structure described by a (possible ad-hoc) markup language.

Unfortunately the pdf version 1.4 is quite old: animations and 3D pictures cannot be embedded, the font format cannot be OpenType, JavaScript programs are not permitted at all, even if they don't modify the document in any way as, for example, a calculator. Ten years ago it was very important to guarantee that the document would always be printed as intended, nowadays screen is slowly replacing paper and animations play a fundamental role: PDF/A-1 is good for paper but less than optimal for "electronic paper".

5 PDF/A-1a in ConTEXt-`mkiv`

Given that it is still under heavy development, ConTEXt-`mkiv` has the opportunity to be developed on two fronts: the "low level" luaTEX (CWEB code and Lua primitives) and the "high level" macros that build the format itself. One of this year results is the implementation of "tagged PDF", the Adobe document markup language for PDF documents, and the development of color macros for the PDF/X specifications. As a consequence, it was possible to use these results to test some real code for producing PDF/A-1a compliant documents. Let's start with an example explained step-by-step.

```
%% Debug
\enabletrackers[backend.format,
                backend.variables]
%% For PDF/A
\setupbackend[
format={pdf/a-1a:2005},
profile={default_cmyk.icc,
         default_rgb.icc,default_gray.icc},
intent={%
ISO coated v2 300\letterpercent\space (ECI)}
]
%% Tagged PDF
%% method=auto ==> default tags by Adobe
\setupstructure[state=start,method=auto]

\definecolor[Cyan][c=1.0,m=0.0,y=0.0,k=0.0]
\starttext
\startchapter[title={Test}]
\startparagraph
\input tufte
%% Some ConTeXt env. are already mapped:
%% colors
\color[red]{OK}
\color[Cyan]{OK}
%% figures
\externalfigure[rgb-icc-sRGB_v4_ICC.jpg]
               [width=0.4\textwidth]
\stopparagraph
%% Natural tables
\bTABLE
\bTR\bTD 1 \eTD \bTD 2 \eTD \eTR
\bTR \bTD[nx=2] 3 \eTD\eTR
\eTABLE
\stopchapter
\stoptext
```

As usual the file is processed with
`#>context test.tex`
and it doesn't hurt to enable some debug information with

```
\enabletrackers[backend.format,
                backend.variables]
```

5.1 Enable the PDF/A-1a

To enable PDF/A-1a we must setup the backend with the appropriate variant of PDF/A. From the very beginning ConTEXt has had a backend system that permits to use almost the same macro-format for different outputs (i.e. DVI and PDF), and with luaTEX this system is increasingly enhanced, as we'll see later on.
With `format={pdf/a-1a:2005}` we select the `1a` variant of PDF/A standard and the label is mandatory because it also puts some default metadata into the output (see `lpdf-pda.xml`; a complete list of formats is currently in `lpdf-fmt.lua` and also as a Lua table `lpdf.formats`).

Next comes the colors part, and we must pay attention here. The key concept is:

every color must be independent of any device.
In a pdf we usually have two sources of colors: the colors specified by the author, e.g something like `\definecolor[orange][r=1.0,g=0.5,b=0.0]` and the images. The 3 most used color spaces `DeviceGray`, `DeviceRGB`, `DeviceCMYK` are device dependent because the reproduction of a color from these color spaces *depends* on the particular output device; but the real output devices are all different due both to the different nature (screen vs. printer, for example) and different technologies (CRT vs. LCD screen, or inkjet vs laser printer, for example). Every device can be classified by means of a *color profile* which maps an input color (rgb, cmyk or gray) to an *independent color space* so that we achieve two goals: such maps assure that each device will correctly reproduce the color, and the independent color space permits to compare colors from different color spaces. With

```
profile={default_cmyk.icc,
         default_rgb.icc,default_gray.icc},
```

we associate all the document colors with the corresponding color profile by mean of a filename (the file `colorprofiles.xml` has a list of predefined profiles). Of course it's wrong to tie a rgb color space to, for example, a cmyk profile, and not all profiles are good too.

There is a second way to specify colors, but it's a bit tricky. We must specify that all the colors *without profile* are intended to be used with a common output profile, i.e. we must impose an *output intent*: this is the meaning of

```
intent={%
ISO coated v2 300\letterpercent\space (ECI)}
```

which is a cmyk profile for coated paper. Note that we are using a name and not a filename to avoid clashing with the values of the `profile` key. By doing so we accept these implicit limitations and color space conversions:

- if the output intent is a cmyk profile then the document can have only cmyk and gray colors;
- if the output intent is a rgb profile then the document can have only rgb and gray colors;
- if the output intent is a gray profile then the document can have only gray colors.

They are reasonable: in general we cannot use a rgb color with a cmyk profile because there are rgb colors without equivalent cmyk ones (that is to say that screens display more colors than printers). We can convert a gray color to rgb or cmyk because usually gray color spaces are a subset of the former (otherwise we have a really poor device). It's not an error if we specify both profiles and output intent: at least if all color spaces have their own profiles, as in the example, then the output intent is simply ignored by a PDF/A compliant pdf reader.

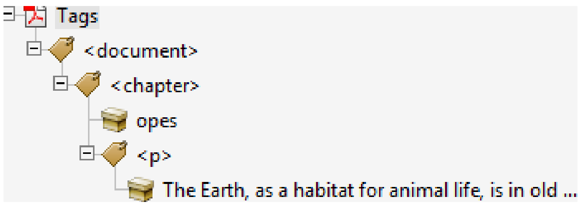
Finally the images: we must be sure that every image has its color profile but as of today ConTEXt-mkiv cannot help here. There are some good programs like MagickWand that are really useful for these tasks.

5.2 Tagged PDF

Next we must enable the tagging system with `\setupstructure[state=start,method=auto]`. ConTEXt-mkiv permits the author to define his own document markup language (the tags used inside the pdf document) but of course we also need the associated TEX macros. This naturally leads to start with a sort of XML document:

```
\setupstructure[state=start,method=none]
\starttext
\startelement[document]
\startelement[chapter]
opes
\startelement[p]\input ward\stopelement \par
\stopelement
\stopelement
\stoptext
```

The internal tag names are `<document>`, `<chapter>` and `<p>` as we see in fig. 1 from Acrobat 9.0, but we still need to put the appropriate typographic elements into the PDF.



```
Tags
  <document>
    <chapter>
      opes
      <p>
        The Earth, as a habitat for animal life, is in old ...
```

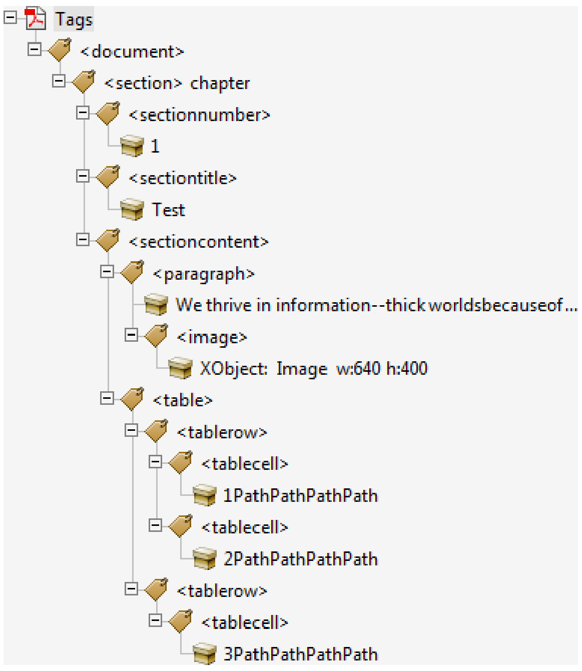
FIGURE 1: The tags structure of a simple document

In the context of PDF/A, a validation program expected the tags as defined by Adobe and this leads to some "syntactic sugar" macros, i.e instead of
`\startelement[chapter]...\stopelement`
it's better to use
`\startchapter[title={Test}]...\stopchapter`
which puts the correct tags and also typesets the chapter title Test as expected.

The complete list of tags can be found in `strc-tag.mkiv` and of course ConTEXt-mkiv permits to redefine the default mapping. In fig. 2 our document shows that ConTEXt-mkiv had already mapped some predefined typographic objects like figures and tables to the appropriate tags.

We can use this mechanism to embed an XML document into a tagged pdf document, which opens quite interesting perspectives, but we can also start from a "structured TEX" document and end into an XML one, and this is more interesting because it's a matter of backend only — and because it's already implemented:

```
\setupbackend[export=yes]
```



```
Tags
  <document>
    <section> chapter
      <sectionnumber>
        1
      <sectiontitle>
        Test
      <sectioncontent>
        <paragraph>
          We thrive in information--thick worldsbecauseof...
          <image>
            XObject: Image w:640 h:400
        <table>
          <tablerow>
            <tablecell>
              1PathPathPathPath
            <tablecell>
              2PathPathPathPath
          <tablerow>
            <tablecell>
              3PathPathPathPath
```

FIGURE 2: The tags structure of a more complex document

```
\setupstructure[state=start,method=none]
\starttext
\startelement[document]
\startelement[chapter][title=Test]
opes
\startelement[p]\input ward\stopelement \par
\stopelement
\stopelement
\stoptext
```

produces a `<tex-file>.export` like this (original XML spaces are not preserved in this listing)

```
<?xml version='1.0' standalone='yes' ?>
<!-- input filename   : test-2              -->
<!-- processing date  : 10/09/10 15:28:48 -->
<!-- context version  : 2010.09.24 11:40   -->
<!-- exporter version : 0.10                -->
<document language='en'
   file='test-2' date='10/09/10 15:29:04'
   context='2010.09.24 11:40'
   version='0.10'>
<chapter title="Test">opes
<p>
The Earth, as a habitat for animal life, is
in old age and has a fatal illness. Several,
in fact. It would be happening whether humans
had ever evolved or not. But our presence is
like the effect of an old-age patient who
smokes many packs of cigarettes per day
------ and we humans are the cigarettes.
</p>
</chapter>
</document>
```

5.3 Fonts and encoding

In the previous subsection we have seen that with simple macro we can have a *valid* (i.e validated by Acrobat 9.0) PDF/A-1a pdf document. We still didn't talk about fonts.

The default fonts used by ConTEXt-mkiv are the OpenType version of LatinModern, and, as of now, they cannot be embedded into PDF/A documents because OpenType isn't supported in version 1.4; this is not a problem because, in essence, ConTEXt-mkiv strips the OpenType part and embeds a valid Type1 or TrueType font. Given an OpenType font, ConTEXt-mkiv is also able to map each glyph to its Unicode id, so even this side is not problematic.

Unfortunately, it's already known that typesetting mathematics with the Computer Modern fonts easily leads to invalid PDF/A documents due to misleading dimensional information of some fonts. As widely noted by C. Beccari, just the simple `$a\not=b$` invalidates the whole document, due the wrong dimension of the `\not` sign. What are the solutions? There are two of them, both unsatisfactory:

1. choose another (valid) math family;
2. make a high resolution (more than 300dpi) bitmap of each invalid formula.

Of course it's possible to edit the fonts, but it's not a general solution: there are copyright limitations and we should subset a modified copy of the font when the original version is different — an error prone situation because modifications of PDF/A-1a document are permitted, and an editor uses the system fonts. The problem remains even if ConTEXt-mkiv can patch the font on the fly. A way out is the complete embedding of the patched fonts, so that the editor uses the document fonts, but it's not a robust solution — some editors can still use the original system fonts. Sometimes just a change of glyph is sufficient:

```
\startluacode
function desc2utf8(desc)
 local us =''
 local plane = 0
 for i,v in pairs(characters.data) do
  if v.description == desc then
   us = v.unicodeslot
   break
  end
 end
 return tex.sprint(tex.ctxcatcodes,
              unicode.utf8.char(us))
end
\stopluacode
\def\N#1{\ctxlua{desc2utf8("#1")}}
\starttext
\startTEXpage
$a\not=b$
$a\N{NOT EQUAL TO}b$ %% Use Unicode names!
\stopTEXpage
\stoptext
```

leads to

$a \neq b$ $a \neq b$

where the first inequality makes an invalid PDF/A-1 while the second does not. But the aesthetic result is very different.

6 Conclusion

The PDF/A1-a is a good standard for document archiving: it's quite complete *Page Description Language*, it relies on Unicode which is also quite good *Character Language* and on Type1 and TrueType as *digital typography formal language*; it has also a good *Document Markup Language*. The binary electronic format and the digital signature for detection and prevention of document modifications complete the picture. The restrictions (e.g. profiles for colors) together with a freely available PDF/A-1a compliant pdf reader lead to an concrete self-contained format.

PDF/A-1a support in ConTEXt-mkiv is still experimental because it needs more tests, but programming in luaTEX is easier than in pdfTEX, and the 1.4 is a well known pdf version. The colors management can be probably improved by permitting to specify a color and its profile for a single object and not for the whole document, as it currently is.

On the other hand, the model of PDF/A-1 is the traditional paper. The exclusion of animations and 3D pictures is questionable and perhaps also the scripting languages should be allowed if they don't change the document.

The ISO standard is not freely available and the PDF/A-1a validators are expensive and complex to implement: this is an obstacle for the diffusion of PDF/A.

In this sense the SVG seems to have more chances than PDF. For example by defining an XML schema that is a subset of full SVG but tailored for document archiving, we automatically gain validation, because free XML validators already exist. From the TEX world we learned that a typographic Turing complete programming language is more powerful than a simple description language, and perhaps SVG can use JavaScript for this — and maybe it will end in a TEX-like language. But even if this scenario will become reality, ConTEXt-mkiv-users can still program typographic tasks as they do today: it will be only a matter of designing a new backend.

7 Notes on References

In section 2, for *digital/micro/macro typography* see Richy and Mittelbach. Unicode Unicode is an example of *Character Language* and Wikipedia Markup is a good starting point for *Markup Languages*; the W3C site XML maintains the specifications for XML. One-way functions are described in OneWayFunction and the pdf reference PDFRef on section "Digital Signatures" shows how it's implemented in PDF.

In section 3, for SVG see the W3C site at SVG; for XSL-FO see Wikipedia at XSLFOWiki and th W3C site at XSLFO. For DocBook see docbook; a complete pdf reference is at PDFRef and for

JavaScript see Wikipedia at javascript.

For section 4, there are some informations on PDF/A-1 at Wikipedia pdfaWiki, at the techdoc at pdfa, and at the digitalpres. Very useful are also the references of C. Beccari's paper at cbpdfa. An interesting use of JavaScript in PDF is calc.

For section 5, the ConTEXt wiki pdfxctxwiki has some terse informations because the code is the ultimate reference. Tagged PDF is described in the new version of `hybrid.pdf` hybrid that is part of "*Proceedings of the 4^{th} ConTEXt meeting*"brejlov, (to be published). For ICC profiles a good starting point is ICCPRofile; font problems are described by C. Beccari in cbpdfa.

References

URL `http://cajun.cs.nott.ac.uk/compsci/epo/papers/volume8/issue2/2point5.pdf`.

URL `http://www.tug.org/interviews/mittelbach.pdf`

URL `http://unicode.org/standard/WhatIsUnicode.html`.

URL `http://en.wikipedia.org/wiki/Markup_language`.

URL `http://www.w3.org/XML/`.

URL `http://en.wikipedia.org/wiki/One-way_function`.

URL `http://www.adobe.com/devnet/pdf/pdf_reference.html`.

URL `http://www.w3.org/TR/2003/REC-SVG11-20030114/`.

URL `http://en.wikipedia.org/wiki/XSL_Formatting_Objects`.

URL `http://www.docbook.org/`.

URL `http://www.w3.org/TR/2006/REC-xsl11-20061205/`.

URL `http://en.wikipedia.org/wiki/JavaScript`.

URL `http://en.wikipedia.org/wiki/PDF/A`.

URL `http://www.pdfa.org/doku.php?id=pdfa:en:techdoc`

URL `www.tug.org/applications/pdftex/calculat.pdf`.

URL `http://www.digitalpreservation.gov/formats/fdd/fdd000125.shtml`.

URL `http://en.wikipedia.org/wiki/ICC_profile`.

URL `http://wiki.contextgarden.net/PDFX`.

URL `http://www.pragma-ade.com/general/manuals/hybrid.pdf`

URL `http://meeting.contextgarden.net/2010/talks/`

URL `http://www.guit.sssup.it/downloads/Beccari_Pdf_archiviabile.pdf`.

▷ Luigi Scarso
`luigi dot scarso at gmail dot com`