Workflow Histories and Image Data with Validation

White Paper 1.84.0.3018

Stefan Vollmar, Andreas Hüsgen,
Michael Sué, Michael May, Roman Krais

Email: vhist@nf.mpg.de

Max-Planck-Institut für
neurologische Forschung
mit Klaus-Joachim-Zülch-Laboratorien
der Max-Planck-Gesellschaft und der
Medizinischen Fakultät der Universität zu Köln
Gleueler Str. 50, 50931 Köln, Germany

VHIST 1.84.0.3018 of Jun 28 2013

Contents

1 Abstract

The VHIST [1] project defines a file format specification that allows to embed arbitrary binary data (with structured meta-information and multiple facilities for validation) as documentation of workflows. It includes a platform independent reference implementation of essential features. The format conforms to the PDF and other open standards, is self-describing and particularly suited as an image or meta-image format in the context of multi-modality and functional imaging. VHIST can be used on top of existing workflows without the need to change major applications. The full format specification and a reference implementation will be subjected to an OpenSource license.

2 Overview and Design Goals

Medical imaging, in particular multi-modality and functional imaging, is well-suited to illustrate the need for tools to document workflows in research environments. However, we do not think that VHIST is by any means limited to this area.

A good level of documentation, e.g. a processing *history* for any image volume in the context of diagnostic purposes or for a meta study, is increasingly a matter of government regulation—apart from being "good scientific practice".

With Multi-Modality Imaging, a typical problem is to provide a complete "log" of processing steps that leads to an image volume. This could initially involve a Positron Emission Tomography (PET) study and an MRI study of the same patient with the goal to combine the PET study's functional with anatomical information from the MRI data, e.g. for surgery planning or neuronavigation. In case of the PET study, documentation should include: patient data, pharmaceutical, acquisition parameters, correction methods, reconstruction and quantification procedures (in research environments, the last three are often in-house developments and only partly supplied by the manufacturer), e.g. [2].

In this example, a complete documentation would also require meta information on the corresponding MRI study and how PET and MRI studies were "fused" [3], on the co-registration method [4] and its parameters, on optimizations for surgery planning (e.g. tumor normalization) [5].

Other examples from a research context in medical imaging include analysis (meta studies) of functional MRI studies. This involves tools for format conversion and packages like FSL [6] and SPM (statistical parametrical mapping) [7].

All these examples have in common that several software packages from different authors (usually using different file formats, logging mechanisms and levels of quality assurance) participate in a multi-step workflow that eventulally results in an image volume. To complicate matters further, many types of studies regularly involve different departments (e.g. departments of neurology, radiology and nuclear medicine), even different institutes, each using different systems and conventions to identify patients.

Some file formats for medical imaging exist that allow for logging of processing steps: future versions of NiFTI [8], DICOM [9], Vista [10] and most notably MINC [11] with NetCDF [12] and future HDF5 support [13]. However, forcing any of these formats on established workflows is often infeasible: conversion problems with potential loss of meta data, missing facilities for validation and being used to file formats that work natively with in-house legacy application causes a low acceptance to change.

To address these issues, we conceived VHIST for usage on top of existing workflows. The general idea behind VHIST is to provide a robust and simple means for documenting steps of a workflow by logging and optionally embedding all relevant information: which files were used, which files were written, what software package in what version was used with what parameters. Ideally, you only have to add one line to an existing batch script; VHIST is intended as a *container format* (so it also is an image or meta-image format)—you can embed arbitrary streams of (binary) data, in particular image files and associated meta-information. The "V" in VHIST is for *validation*, one of the format/concept's main features:

in each step of a workflow, a meta-log, associated files and an automatically generated summary in XML are embedded and/or "finger-printed" (MD5 [14]).

Saving structured content for each workflow step (as XML summaries) facilitates automated processing, e.g. for reporting the differences in the processing histories of two image volumes. This can lead to reports covering several parallel processing branches if the VHIST files were set up to contain other VHIST data (as embedded files).

VHIST-files are stacks of *sections*, each section can be validated independently. Adding a new section to a VHIST-file does not modify any existing file contents, it is an *append-only* operation. This is one reason we have decided against popular archiving formats which can also be used as container formats, e.g. zip-archives [15] and jar-files [16]. VHIST encodes all text entries in Unicode (UTF-8 [17]).

The VHIST format conforms to the PDF-1.5 specification: a VHIST-file can be opened with any PDF browser, embedded data maps to "embedded files". However, for extracting and/or processing of data, no knowledge of the PDF standard is required: a major design goal of the VHIST format is the ability to extract all relevant information with basic programming techniques, a 25-line program written in the Python [18] programming language (embedded in each VHIST file) is sufficient.

The format is self-describing: you can read all required instructions for usage with a "more" command. A VHIST file has a title page which can contain information about the institution where the file was "assembled". In addition, it might contain a *legal notice* about intellectual properties, permissions and conditions regarding scientific or other use of the embedded data.

VHIST opens up interesting possibilities to combine existing meta-information of images and workflows (i.e. log files of acquisition or processing procedures) with image data formats which only have limited support for meta-information. Other applications might include structured information such as provided by XCEDE [19].

3 Examples

Figure 1: Left hand: screenshot of user supplied text (excerpt of PDF presentation, this page is presented to the user on opening the VHIST file), right: same text as source file with Wiki-like markup.

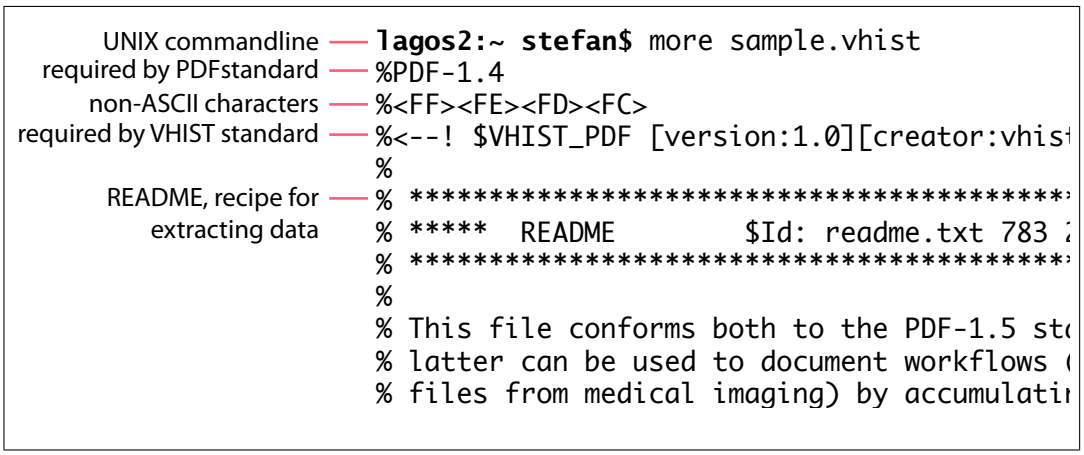

Figure 2: VHIST file unfiltered. After the initial "magic" required by the PDF standard, the VHIST file contains a brief format description with sample code to extract data. The non-ASCII data signals algorithms for heuristic file type recognition that this file should be treated as binary data when transferring it.

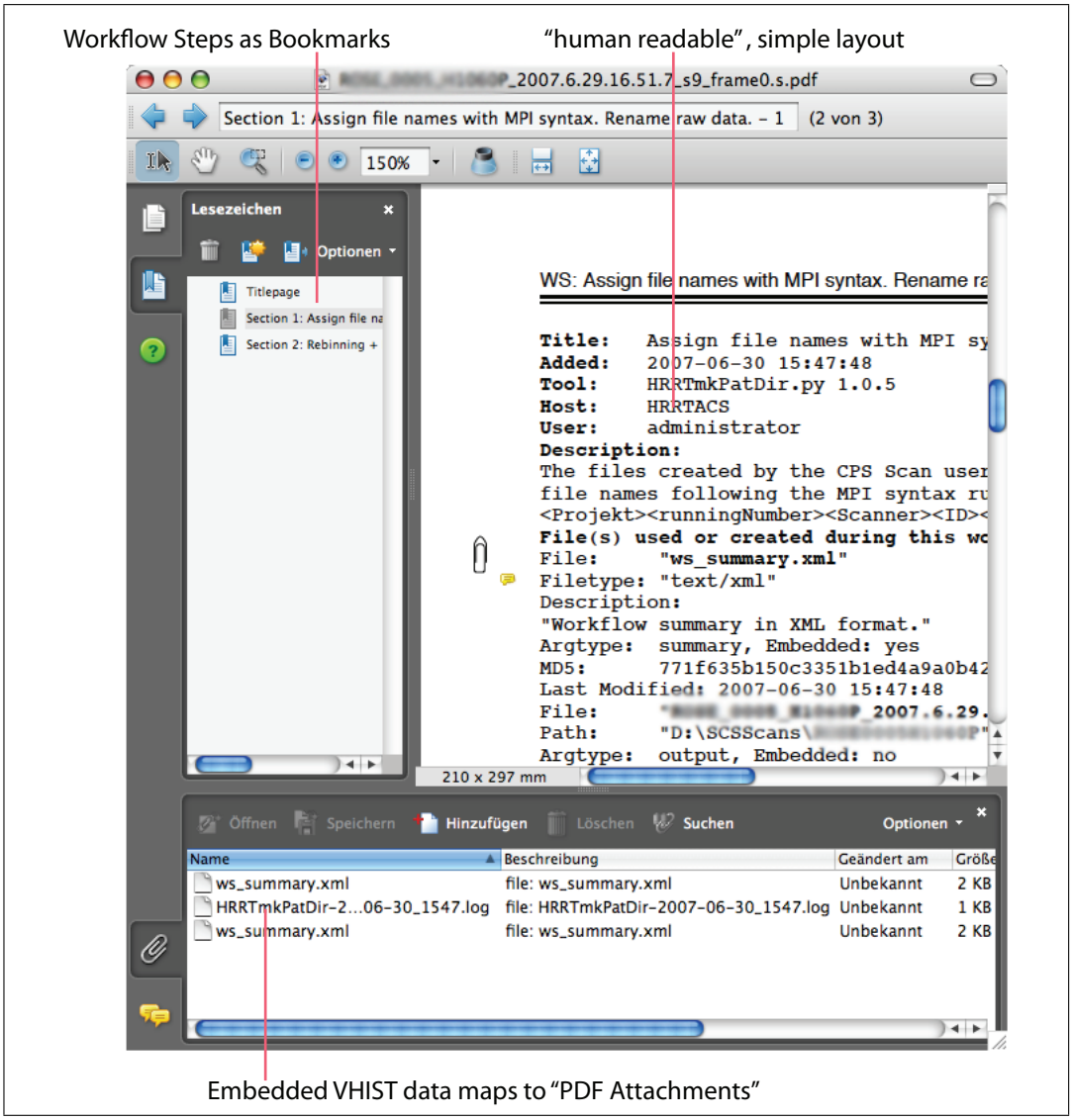

Figure 3: Screenshot of a VHIST file opened in Adobe Acrobat Pro 8. The PDF browser has been configured to display bookmarks (left) and embedded files (bottom).

```
lagos2:~ stefan$ strings sample.vhist | grep \$VHIST_EMBEDDEDFILE_BEGIN
%<--! $VHIST_EMBEDDEDFILE_BEGIN
[filetype:][filename:K:\\Cologne\\Log\\2007\\HRRTmkPatDir-
2007-06-30_1547.log][desc:][comment:][compression:flate][filesize:941][cf
ilesize:423][blocksize:443][offset:8][md5file:586bd37ff095969f05ae872442
c0154][md5cfile:76b87acc97deb0b1199d3e1d18784482]-->
%<--! $VHIST_EMBEDDEDFILE_BEGIN
[filetype:text/xml][filename:ws_summary.xml][desc:Workflow summary in XML
format.][comment:][compression:flate][filesize:1997][cfilesize:842][block
size:862][offset:8][md5file:771f635b150c3351b1ed4a9a0b429d9d][md5cfile:66
109ddb2c45307422e37d2b7e2c2489]-->
%<--! $VHIST_EMBEDDEDFILE_BEGIN
[filetype:text/xml][filename:ws_summary.xml][desc:Workflow summary in XML
format.][comment:][compression:flate][filesize:2006][cfilesize:744][block
```

Figure 4: VHIST was conceived to allow for "pragmatic" extraction of data with time-honoured commandline tools—in parallel to more modern methods including validation during the extraction process.

```
#!/usr/bin/env python
import os, re, zlib
histfilename = os.sys.argv[1]
histfile = open(histfilename, "rb")
data = histfile.read()
histfile.close()
curpos, nextfile = 0L, 0
regex = re.compile(r"<--! \$VHIST_EMBEDDEDFILE_BEGIN(.*?)"
                   r"\[filename:(?P<fn>([^\\\]]|(\\\])|(\\))|(\\\])|(\\n))+?)\](.*?)"
                   r"\[compression:(?P<cp>.*?)\](.*?)"
                   r"\[filesize:(?P<fs>[0-9]+?)\](.*?)"
                   r"\[cfilesize:(?P<cfs>[0-9]*?)\](.*?)"
                   r"\[offset:(?P<os>[0-9]*?)\](.*?)"
                   r"(.*?)-->", re.DOTALL)
found = regex.search(data, curpos)
while found is not None:
    filename, compression, bytesize, cbytesize, filepos = found.group("fn"), \
        found.group("cp"), int(found.group("fs")), found.group("cfs") == "" or \
            int(found.group("cfs")), found.end() + int(found.group("os"))
    filename = filename.replace("\\n", "\n").replace("\\]", "]").replace("\\\\", "\\")
    outfile = open("[%d]_%s" % (nextfile, filename), "wb")
    if compression == "none": outfile.write(data[filepos:filepos + bytesize])
    elif compression == "flate":
        outfile.write(zlib.decompress(data[filepos:filepos + cbytesize]))
        bytesize = cbytesize
    else: print "unknown compression method!"
    outfile.close()
    nextfile, curpos = nextfile + 1, filepos + bytesize
    found = regex.search(data, curpos)
```

Figure 5: Minimum standalone program written in the Python programming language, suitable for extracting all embedded files (or data) from a VHIST file, takes care of decompression ("inflation") if required. This listing is embedded in each VHIST file as a "recipe" for extracting data.

4 The VHIST Format

4.1 Introduction

VHIST files are stacks of **4.5 Sections** which can be validated independently and usually refer to one **4.8 Workflow Step**. Sections are appended at the end of an existing file, so no previous data is changed. This is related to incremental writing of PDF [20] files.

Each section contains **4.6 Embedded Data**, i.e. one or more "embedded files" from a PDF browser's point of view. The first (and possibly only) data stream ("file") is an automatically generated **4.7 XML Summary** which summarizes the workflow step in a structured way suitable for automated processing and recreation of damaged VHIST files.

In addition, a **4.9 Workflow Presentation (PDF)** is generated using a subset of PDF and some pragmatic layout constraints to facilitate automatic line and page breaks. This presentation is intended for comfortable viewing of workflow steps with a PDF browser and extraction of embedded data without special VHIST tools, but not for automated processing.

The VHIST format uses a certain degree of redundancy to improve robustness: XML

and PDF representation of a workflow, **BEGIN** and **END** markers comprising the same information, additional MD5 checksums. It is even possible to fully recover a VHIST file if all meta-information have accidentally been stripped by an external tool (as long as the embedded files are still available).

VHIST was designed to allow for efficient and simple processing by providing meta-information for parsing and **5.2 Validation**. By design, no knowledge of PDF is necessary for processing this information, see **5.3 Extracting Data**.

But some constraints are imposed on the structure of a file if it is intended to conform to the PDF specification [20] (to avoid confusion: VHIST still allows to store arbitrary binary data). In particular, the PDF standard is explicit about the first and trailing bytes of a file. It also poses constraints about data immediately preceding and following embedded streams. These constrains are reflected in the definition of the **4.4 VHIST marker**: it allows to embed meta-information as PDF comments (which are ignored by PDF browsers) and allows for offset-information for flexible positioning which does not interfere with PDF's constraints.

This chapter is not a full specification of the VHIST format: it is incomplete with regards to our proposed usage of PDF objects. For pragmatic reasons, we limit VHIST to only use a subset of PDF's functionality, e.g. only use a fixed number of fonts. This is described in detail in the *The* VHIST *specification* [1].

4.2 Typographical Notes

In this document, PDF-specific parts of VHIST listings are set in this way:

[...] <</Type /EmbeddedFile /DL 2043 /Filter /FlateDecode [...]

to distinguish them from VHIST meta-information (ignored by PDF browsers):

%<--!·**$VHIST_EMBEDDEDFILE_BEGIN**·¬

Please note that "·" denotes a <space> character and "¬" is used to indicate a line continuation, i.e. the line is broken to improve readability not because it ends with an end of line marker. If no line continuation marker "¬" is used, the linebreak reflects the presence of a UNIX lineending character (0x0a). This is an example of a comment line (set in a smaller font):

[...] 904 Bytes of compressed data (2043 Bytes uncompressed) [...]

Parts of VHIST meta-information that corresponds to tags and attributes in XML streams has been set in bold text.

4.3 Conventions and Data types

4.3.1 Line endings

vhifollow the UNIX style: a line ends in one character (0x0a).

The following definitions only affect the parts of a VHIST file which are not PDF specific.

4.3.2 attrText <maxlength>

A text string encoded in UTF-8 [17] format (unicode, downward compatible with ASCII). In the context of **4.4 VHIST marker**, the characters "\" and "]" have a special meaning, therefore must be escaped as "\\" and "\]", resp. Further, newline characters (Unix: 0x0a, MacOS Classic: 0x0d, Windows: 0x0d 0x0a) are not permitted within a **4.4 VHIST marker**. Therefore, newlines in attributes must be escaped using "\n". VHIST does not differentiate between the different flavors of newline representations, since all are semantically equal in ordinary text. This way, tools extracting VHIST markers can easily identify newlines and convert them into the system specific representation. In addition, text entries in attributes

must not contain \0 characters (C end-of-string markers). If specified, **<maxlength>** limits the size of the text string to that many bytes. CAVEAT: as UTF-8 needs more than one byte to encode characters outside the range of 7-bit ASCII, the size constraint of **<maxlength>** bytes can limit a string to less than **<maxlength>** characters.

4.3.3 attrUINT

An unsigned integer, must be specified as decimal value, may have leading zeros.

4.3.4 attrMD5

A text string with an MD5 checksum [14]. Should be in lower-case hex notation and always has a size of 32 bytes, e.g. the 32 characters `2499fa16c40e8aba1fd58e99a59b25fd`.

4.3.5 Block/file sizes and offsets

are given in bytes: decimal numbers, no units.

4.3.6 Integer numbers

are not required to have a fixed length (padding), but they may have leading zeros to facilitate implementation. Integer numbers must be specified as decimal values.

4.3.7 Empty attributes

In some special cases, it is legal to leave **4.3.3 attrUINT** or **4.3.4 attrMD5** attributes empty. In these cases, it is explicitly mentioned in the attribute's description. It is, however, not allowed to omit the complete attribute.

4.4 VHIST marker

We tried to keep the marker's syntax as simple as possible. The term "marker" refers to an important property of VHIST files: if more sophisticated methods are not required (or *fail* because a file was damaged), embedded data can be retrieved by locating the markers.

Marker tags with associated attributes are defined in **4.5 Sections** and **4.6 Embedded Data**. The general structure of a marker is given in Fig. 6:

```
%<--!·$VHIST_TAG· [attribute1:data1][attribute2:data2]-->
```

Figure 6: Structure of a VHIST marker. "%" starts a PDF comment. Parts of the marker that conceptually correspond to tags and attributes in XML streams have been set in bold text.

4.4.1 Meta-information as PDF comments

VHIST incorporates meta-information as PDF comments. PDF [20] requires comments to start with a "%" character and end with an end of line marker (use character 0x0a.) Moreover, VHIST requires PDF comments comprising VHIST meta-information to be positioned at the beginning of a line, i.e. the line begins with "%" immediately succeeded by the VHIST marker.

4.4.2 Syntax

A marker starts with "<--!·" followed by one <space> character and the marker's "tag". The marker's attributes are key-value pairs, separated with a colon (no white space) and delimited by square brackets. The first attribute is separated from the marker's tag by one <space>. The marker ends with "-->" immediately after the last attribute's closing square bracket. A tag starts with **$VHIST_** and only uses characters from this set: ['A'...'Z', '_'] (upper-case ASCII, with '_').

4.4.3 Encoding of Attributes

Attributes are key-value pairs, keys must use characters from this set: ['a'..'z', '-'] (lowercase ASCII, with "-"). Attribute values must be formatted as described in **4.3 Conventions and Data types**.

4.4.4 VHIST markers and Offsets

The "<--!·" and "-->" strings are reminiscent of XML comments. Offsets refer to either the "<" (keyword: **left**) or ">" (keyword: **offset**) character of a marker, i.e. an **offset** of zero refers to the ">" character's position.

4.4.5 Order of Attributes

The order of attributes is mandatory.

4.5 Sections

A VHIST file is a stack of sections, i.e. every byte of the file belongs to one section. A section is that part of a file which is written when adding data for one **4.8 Workflow Step**.

Each section can be validated independently. In order to achieve that, the section's **4.4 VHIST marker** needs to specify the extent of the section relative to the marker's file position. Adding a section to a VHIST file must not change previous sections.

```
%<--!·$VHIST_SECTION·¬
[version:VHIST-1.00]¬
[creator:vhistadd v 0.11 of May 26 2006]¬
[title:Rebinding (HRRT)]¬
[left:11027] [size:000000011782] [index:4]¬
[md5section:56a9c5934c20b282220c70e3db9131ef]¬
[previousmd5:7591ab861d09d066cae992935c44e40a]¬
[previousmarker:27805]-->
```

Figure 7: A section's VHIST marker.

The general structure of a marker is given in Fig. 7. The following attributes are mandatory and must appear in this order:

4.5.1 Attribute: `version` [4.3.2 attrText <maxlength>]

The version of the VHIST standard. For the current specification, must be: "VHIST-1.00".

4.5.2 Attribute: `creator` [4.3.2 attrText <maxlength>]

Name and version of the program which appended this section to a VHIST file.

4.5.3 Attribute: `title` [4.3.2 attrText <maxlength>]

User specified title of the section.

4.5.4 Attribute: `left` [4.3.3 attrUINT]

The section starts **left** bytes from the section markers start, i.e. this is an offset from the "<" character (the section starts left of the section marker). A value of zero refers to the "<" character. This allows a section to start with some PDF "magics".

4.5.5 Attribute: `size` [4.3.3 attrUINT]

The total number of bytes in this section. The section starts left of the marker's start position ("<") and extends beyond the marker's right boundary (">"). This value is likely to be preceded by zeros for padding reasons, since the size of the section is usually not known during its creation and depends on the value of **size**.

4.5.6 Attribute: `index` [4.3.3 attrUINT]

Sections are indexed in ascending order (“age”), the very first section’s index is one.

4.5.7 Attribute: `md5section` [4.3.4 attrMD5]

The MD5 message digest (“electronic fingerprint”) [14] of the entire section in lower-case hex notation (32 bytes). As this message digest is included in this fingerprint, some additional processing is necessary for validation: extract the entire section, copy the MD5 checksum, replace the MD5 checksum with 32 times the character “0” (zero), calculate the MD5 checksum of the modified section, compare with the original MD5 checksum.

4.5.8 Attribute: `previousmd5` [4.3.4 attrMD5]

The previous section’s **`md5section`** attribute. As validation of each section is independent of other sections, exhaustive validation requires this attribute to make sure that all sections are in place and in the correct order. The first section’s **`previousmd5`** attribute must be left empty.

4.5.9 Attribute: `previousmarker` [4.3.3 attrUINT]

The offset from the marker’s start position (“<”) to the previous section marker’s start position. This attribute is useful to scan a VHIST file efficiently in order to generate a table of contents (combined with some basic validation). The first sections’s **`previousmarker`** attribute must be left empty.

4.6 Embedded Data

Each of the 4.5 Sections contains one or more embedded data streams which can be used for storing binary data, e.g. the contents of files, or structured information (XML strings).

```
24 0 obj
<< /Type /EmbeddedFile /DL 2043 /Filter /FlateDecode
/Params << /Size 904 /ModDate (D:20060622064141) >>
/Length 2043 >>

%<--!·$VHIST_EMBEDDEDFILE_BEGIN·¬
[filetype:text/ini-format] [filename:SomeExampleFile.ini]¬
[desc:Contains HRRT site-specific parameters]¬
[comment:Test configuration]¬
[compression:flate] [filesize:2043] [cfilesize:904] ¬
[blocksize:924] [offset:8]¬
[md5file:254df8f02e7e232d8a472489d87b3852]¬
[md5cfile:fa61a6d08d0dc0bdea8246065d807d92] -->

stream
... 904 Bytes of compressed data (2043 Bytes uncompressed) ...
endstream

%<--!·$VHIST_EMBEDDEDFILE_END·¬
[filetype:text/ini-format] [filename:SomeExampleFile.ini]¬
[desc:Contains HRRT site-specific parameters]¬
[comment:Test configuration]¬
[compression:flate] [filesize:2043] [cfilesize:904] ¬
[blocksize:924] [offset:8]¬
[md5file:254df8f02e7e232d8a472489d87b3852]¬
[md5cfile:fa61a6d08d0dc0bdea8246065d807d92] -->
```

Figure 8: In this excerpt from a VHIST file, a file with original name `SomeExampleFile.ini` has been embedded with a fixed offset of 8 bytes after the marker’s last character (“>”, thus skipping the “stream” keyword and two line endings). The first five lines are PDF specific.

Embedded data maps to "embedded files" from a PDF browser's point of view. As can be seen from the example in Fig. 8, the embedded data must be preceded and immediately followed by PDF keywords stream and endstream, resp.

4.6.1 Positioning EMBEDDEDFILE markers

A **4.4 VHIST marker** with tag **`$VHIST_EMBEDDEDFILE_BEGIN`** is positioned before the stream, a corresponding marker with tag **`$VHIST_EMBEDDEDFILE_END`** should follow the endstream keyword. Please note that the VHIST markers must end with a lineend character, also that a lineend must follow the endstream keyword.

4.6.2 EMBEDDEDFILE markers and attributes

Both VHIST markers (**`EMBEDDEDFILE_BEGIN`** and **`EMBEDDEDFILE_END`**) have the same set of attributes with identical values, so VHIST files can easily be parsed front-to-back and back-to-front. The following attributes are mandatory.

4.6.3 Attribute: `filetype` [4.3.2 attrText <maxlength>] <255>

A text string with a text-description of the file's datatype (similar to a MIME type [21]), e.g. `binary/Analyze-Header`. See **4.10.2 Comments on Attribute `filetype`** for a discussion.

4.6.4 Attribute: `filename` [4.3.2 attrText <maxlength>] <255>

A text string with the name of the embedded datastream. If the embedded data was read from an external file this attribute should contain that file's original name (without path; the full path is stored in the XML summary). To avoid platform-dependent problems, we recommend to only use 7-Bit ASCII for filenames and avoid newlines, slashes, backslashes and further characters, which may be invalid within filenames on various platforms. When extracting embedded files, it is important to keep in mind that this attribute is not necessarily unique, i.e. other embedded files might have the same name.

4.6.5 Attribute: `desc` [4.3.2 attrText <maxlength>]

A text string with a description of the embedded file.

4.6.6 Attribute: `comment` [4.3.2 attrText <maxlength>]

An text string containing additional comments referring to the embedded file.

4.6.7 Attribute: `compression` [4.3.2 attrText <maxlength>]

Either `none` or `flate`. The latter refers to the lossless flate compression method. It is based on the public-domain zlib/deflate compression method [22], which is derived from the Lempel-Ziv algorithm [23]. This type of compression was chosen because it is widely available and "native" to the PDF standard: PDF browsers can uncompress ("de-flate") data when extracting embedded files.

4.6.8 Attribute: `filesize` [4.3.3 attrUINT]

The original file's size, or, generally, the original (uncompressed) size of the embedded data in bytes. If compression is used, the size of the embedded data is usually smaller than the original size (this is the general idea of compression).

4.6.9 Attribute: `cfilesize` [4.3.3 attrUINT]

The size of the embedded data (after compression). This should correspond to the number of bytes between the stream and endstream keywords (minus the two line endings which are not counted). If the file is not compressed, i.e. **`compression`** is `none`, **`cfilesize`** must be empty.

4.6.10 Attribute: `blocksize` [4.3.3 attrUINT]

The number of bytes between the two VHIST markers (**`EMBEDDEDFILE_BEGIN`** and **`END`**, starting with the first byte after the **`BEGIN`** marker's ">" character up to and including the byte preceding the **`END`** marker's "<" character. In PDF-1.5 this is equal to:

```
sizeof('\n') + sizeof('stream\n') + cfilesize + sizeof('\n') +
sizeof('endstream\n') + sizeof('%') = cfilesize + 20
```

and useful for parsing a VHIST file back-to-front.

4.6.11 Attribute: `offset` [4.3.3 attrUINT]

The number of bytes between the **`EMBEDDEDFILE_BEGIN`** marker's end and the first byte of the data stream (a value of zero refers to the character after the marker's '>' character, i.e. the stream begins immediately after the '>' character). In PDF-1.5 this is equal to:

```
sizeof('\n') + sizeof('stream\n') = 8
```

4.6.12 Attribute: `md5file` [4.3.4 attrMD5]

The MD5 message digest ("electronic fingerprint") [14] of the original file (or uncompressed data) in lower-case hex notation (32 bytes).

4.6.13 Attribute: `md5cfile` [4.3.4 attrMD5]

The MD5 message digest ("electronic fingerprint") [14] of the compressed file (or compressed data) in lower-case hex notation (32 bytes). If **`compression`** is `none`, i.e. the file is not compressed, **`md5cfile`** must be empty.

4.7 XML Summary

All information stored for a particular **4.8 Workflow Step** (e.g. **4.6 Embedded Data**, title or description of the workflow step, optional key-value pairs) is recorded as structured data (XML), with a view to automated processing. This summary is generated automatically and embedded as a file.

4.8 Workflow Step

Refers to a particular stage of a processing history where a particular tool (program) acts on some input data (parameters and files) and creates one or more files. Each workflow step is associated with one of theVHIST file's **4.5 Sections**.

4.9 Workflow Presentation (PDF)

As can been from Fig. **3**, we have implented a simple and pragmatic "human readable" overview of all information stored in a particular **4.8 Workflow Step**. A single mono-spaced font family is used (natively available with all PDF browsers), so layout control (line and page breaks) is straightforward. Although it is possible to parse this PDF representation of a workflow step, for automated processing of VHIST files the **4.7 XML Summary** should be used.

4.10 Comments on specific Design Decisions

4.10.1 The syntax of VHIST markers

A resemblance to XML comments has been chosen, so we could define start and end of a marker in an analogue way (using the '<', '>' characters). In addition, we chose the format of "comment" to convey that the marker's content (the list of attributes) does not conform

to XML. The very nature of file formats that support embedding binary data prohibits the option to use XML throughout the whole file. The syntax of the marker's attributes was chosen to allow for convenient parsing, e.g. using *regular expressions* as shown in examples Fig. 4 and Fig. 5). In this context, using XML would not have been an advantage (the added flexibility is not needed here), but rather made parsing unnecessarily difficult.

4.10.2 Comments on Attribute `filetype`

The attribute **`filetype`** has been added for two reasons. (a) In medical imaging, file extensions are far from conclusive as they can specify quite different file types, e.g. `file1.hdr` can be the header file of an image volume in Interfile format [24] (text format, key-value pairs), while `file2.hdr` might be the binary header of an Analyze [25] file. Providing type information of this kind (*text* vs *binary*, *Interfile* vs *Analyze*) can serve for additional plausibility checks and provide useful hints on how to open/process certain embedded data streams. (b) Automated processing of VHIST files can be made more efficient, e.g. a VHIST file with two co-registered image volumes might contain a third file which specifies a transformation matrix (or, indeed, all information related to a fully-automatic co-registration task). For automated processing (which, in this example, requires extraction of the transformation matrix at some stage) the **`filetype`** attribute allows to immediately select the appropriate embedded file from an extended listing of embedded data, here: `xml/Vinci-CoReg-1.0`. The syntax suggested here is losely inspired by MIME Content-Type conventions [21].

5 Processing VHIST data

5.1 Adding Data

VHIST files consist of **4.5 Sections**, the smallest unit for adding is one section without any embedded files. By design, adding is a strict *append-only* operation (compatible with PDF standard) and thus the safest way to avoid corruption due to processing errors while adding data: previously valid data remains valid, allowing for section-wide **5.2 Validation** and improving robustness of the format. A section can be added using the reference implementation's **6.2 vhistadd** tool.

5.2 Validation

Validation of data (MD5) is possible at several levels in VHIST files. In general, individual **4.5 Sections** can be validated: so data corruption can be pinpointed (at least) with section granularity. In addition, embedded data (files) can be validated in compressed and uncompressed form, also checksums of referenced files are available for validation.

5.3 Extracting Data

Several methods exist to extract data from a VHIST file: (a) use a PDF browser (extract PDF attachments), or (b) use a pragmatic approach with venerable commandline utilities (see Fig. 4), or (c) the minimum standalone Python listing from Fig. 5, or (d) the **6.3 vhistxl** tool provided with the **6 Reference Implementation**.

5.4 Mapping Histories

Given two image volumes with a VHIST-based history, it is possible to iterate over the embedded **4.7 XML Summary** files in each VHIST file and report the differences in their processing history. This can lead to reports covering several parallel processing branches if the VHIST files were set up to contain other VHIST data (as embedded files).

6 Reference Implementation

In addition to the VHIST specification, a reference implementation with an array of tools for tasks comprising file generation, extraction and validation is provided. These tools are available under an open source license on all major operating systems. Current development and automated testing focuses on the platforms MS Windows, Linux, MacOS X and Solaris.

6.1 Architecture

The tools in the reference implemenation have been developed with compatibility, portability and usability in mind. We think that the Python programming language [18] is a good basis to achieve these goals as it is maintained by an extensive community of developers and used in a lot of open source and commercial applications, both as scripting and as a standalone language. Furthermore, it is preinstalled on most platforms including Linux, Mac OS X and most derivatives of Unix. On Windows NT based systems, which do not provide preinstalled Python implementations, the VHIST reference implementation can be supplied as a set of self containing executables including the Python interpreter. Care was taken that no additional libraries are required to use VHIST and we made sure that a Python 2.3 distribution should be sufficient.

One important goal of VHIST is the possibility to generate VHIST documentation automatically from within existing workflows. To facilitate this, all central components of the VHIST reference implementation provide commandline interfaces thus allowing to integrate VHIST tools into batch files, shell scripts, makefiles and all programming and scripting languages, which support the creation of subprocesses. Calling VHIST tools manually on remote machines over ssh or on machines without a graphical userinterface is also supported.

6.2 vhistadd

We developed `vhistadd` as a reference tool for creating new VHIST files and extending existing files by appending additional **4.5 Sections**. Its commandline interface has been developed for easy integration into existing scripts that control one or more workflow steps. **6.4 vhistez** complements `vhistadd` with a graphical user interface for configuration. `vhistadd` has been written entirely in Python, using core libraries only.

6.3 vhistxl

The commandline tool `vhistxl` can be used for extracting and validating information contained in existing VHIST files. It supports fast listing of all **4.5 Sections** and **4.6 Embedded Data** inside a VHIST file as well as validation and extraction, keeping track of embedded files with identical filenames.

6.4 vhistez

The `vhistez` tool (C++ and Qt 4.3 [26]) has been designed for easy and comfortable creation of commandline calls for use with **6.2 vhistadd**. It is an orthogonal effort to increase acceptance for sites that have little scripting experience but might also be useful to proficient users.

7 Discussion

We think that the VHIST project provides the means to fill an important gap in documenting workflows in research environments, in particular in the context of medical imaging: here,

using file formats with enhanced capabilities to store meta information of processing histories or rewriting existing tools for improved quality assurance is often not an option. First attempts to integrate VHIST in complex medical workflows have been encouraging and we hope that researchers in areas outside of medical imaging will also find the pragmatic nature of the VHIST format appealing.

8 Acknowledgements

A number of colleagues have contributed ideas and criticism to the VHIST project. We would like to thank Karl Herholz, Mauritius Hoevels and Marc Tittgemeyer for fruitful discussions and support.

References

[1] S. Vollmar, A. Hüsgen, M. Sué, The VHIST Homepage (2007).
URL http://www.nf.mpg.de/vhist

[2] S. Vollmar, C. Michel, J. Treffert, D. Newport, M. Casey, C. Knöss, K. Wienhard, X. Liu, M. Defrise, W.-D. Heiss, HeinzelCluster: accelerated reconstruction for FORE and OSEM3D, Physics in Medicine and Biology 47 (2002) 2651–2658.

[3] S. Vollmar, J. Cizek, M. Sue, J. Klein, A. H. Jacobs, K. Herholz, VINCI -Volume Imaging in Neurological Research, Co-Registration and ROIs included, Forschung und wissenschaftliches Rechnen 2003 (Kremer K, Macho V, eds), Göttingen: GWDG (2004) 115–131.

[4] J. Cizek, K. Herholz, S. Vollmar, R. Schrader, J. Klein, W. D. Heiss, Fast and robust registration of PET and MR images of human brain, NeuroImage 1 (22) (2004) 434–42.

[5] S. Vollmar, J. Hampl, L. Kracht, K. Herholz, Integration of Functional Data (PET) into Brain Surgery Planning und Neuronavigation, Advances in Medical Engineering, Springer Proceedings in Physics 114 (2007) 98–103.

[6] S. Smith, M. Jenkinson, M. Woolrich, C. Beckmann, T. Behrens, H. Johansen-Berg, P. Bannister, M. D. Luca, I. Drobnjak, D. Flitney, R. Niazy, J. Saunders, J. Vickers, Y. Zhang, N. D. Stefano, J. Brady, P. Matthews, Advances in functional and structural MR image analysis and implementation as FSL, NeuroImage 23(S1) (2004) 208–219.
URL http://www.fmrib.ox.ac.uk/fsl

[7] Wellcome Department of Imaging Neuroscience, SPM.
URL http://www.fil.ion.ucl.ac.uk/spm

[8] NIfTI: Neuroimaging Informatics Technology Initiative.
URL http://nifti.nimh.nih.gov/nifti-1

[9] Digital Imaging and Communications in Medicine (DICOM).
URL http://medical.nema.org

[10] G. Wollny, M. Tittgemeyer, Some notes on the vista package.
URL http://mia.sourceforge.net/vista.html

[11] P. Neelin, The MINC web page.
URL http://www.bic.mni.mcgill.ca/software/minc

[12] NetCDF (network Common Data Form).
URL http://www.unidata.ucar.edu/software/netcdf/

[13] The HDF5 Homepage.
URL http://hdfgroup.com/HDF5/

[14] R. Rivest, The MD5 Message-Digest Algorithm, RFC 1321.

[15] ZIP (file format).
URL http://en.wikipedia.org/wiki/ZIP_(file_format)

[16] Packaging Programs in JAR Files.
URL http://java.sun.com/docs/books/tutorial/deployment/jar/

[17] W3C, Unicode (UTF-8, UTF-16).
URL http://www.ietf.org/rfc/rfc2781.txt

[18] Python Programming Language Official Website.
URL http://www.python.org

[19] XCEDE (XML-Based Clinical Experiment Data Exchange Schema).
URL http://nbirn.net/Resources/Downloads/XCEDE

[20] Adobe Corp., Adobe Systems Incorporated, PDF Reference, fourth edition, Adobe Portable Document Format Version 1.5.
URL http://www.adobe.com/devnet/pdf/pdf_reference.html

[21] W3C, RFC 2046, Multipurpose Internet Mail Extensions (MIME) Part Two: Media.
URL http://tools.ietf.org/html/rfc2046

[22] L. P. Deutsch, DEFLATE Compressed Data Format version 1.3.
URL http://www.ietf.org/rfc/rfc1951.txt

[23] J. Ziv, A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory IT-23 (3) (1977) 337–343.

[24] T. Cradduck, D. Bailey, B. Hutton, F. Deconinck, E. B. Sokole, H. Bergmann, U. Noelpp, A standard protocol for the exchange of nuclear medicine image files, Nucl Med Commun 10 (1989) 703–713.

[25] Mayo Foundation, Analyze 7.5 File Format.
URL http://www.mayo.edu/bir/PDF/ANALYZE75.pdf

[26] Qt-Trolltech.
URL http://trolltech.com/products/qt