NOT WELL-FORMED OR INVALID. NOW WHAT?

Towards a formalized workflow for format validation error treatment

Micky Lindlar
TIB Leibniz Information Centre for Science and Technology
Germany
michelle.lindlar@tib.eu
0000-0003-3709-5608

Abstract – File format validation – we all use it and we all run into problems when files do not validate. Though a core process within digital preservation practice, little progress has been made in shared documentation and discussion of processes used to treat file format validation errors. This paper aims to close that gap. A basic workflow for handling validation errors is proposed and visualized, and in a second step tested against two TIFF and two PDF validation errors of varying severity. Observations made are fed back into the workflow diagram. The outcome shall provide a first step towards shared digital preservation practice in the currently largely neglected field of method formalization for file format validation error treatment.

Keywords – file format validation; process formalization; error handling

Conference Topics – We're all in this Together; From Theory to Practice

I. INTRODUCTION

(Digital) interpretability, i.e., correct rendering of digital objects, is one of the core tasks of digital preservation. Checking if files open in a reader is one method to check correct rendering, however, this is a time-intensive process and can only be achieved perfectly if we know what the original is supposed to look like. Even in textual objects, malformed formulas or tables might easily be missed during visual inspection [1],[2]. File format identification and file format validation therefore serve as standard processes to check for a file's structural and syntactical intactness and they are embedded processes in all major end-to-end digital preservation systems [3]. It is thus safe to say that most digital preservation practitioners should be familiar with the tasks.

Since file format identification is based on short pattern recognition such as "magic number" or byte sequence checking, it is a good first indicator of a digital object's file format, but it is by no means proof that the file can actually be opened. The most reliable method to ensure the renderability of a digital object is file format validation, which checks the digital object's internal syntax against the rules outlined in the file format standard or description. Since the development of a validator not only depends on the availability of a file format description, but is also significantly more resource-intensive than identifying and capturing a file format signature pattern, validators are currently only available for a handful of file format families. Those file formats that do have validators available are often also found in "recommended" or "preferred file format lists" [4].

iPRES 2023: The 19th International Conference on Digital Preservation, Champaign-Urbana, IL, US.
Copyright held by the author(s). The text of this paper is published under a CC BY-SA license (https://creativecommons.org/licenses/by/4.0/).
https://hdl.handle.net/2142/121092

Benefits of a formalized methodology for validation troubleshooting are threefold:

(1) "A picture is worth a thousand words" – a workflow graphic can enable more effective communication about specific errors amongst practitioners. Furthermore, it can help those just starting out to understand the process better.

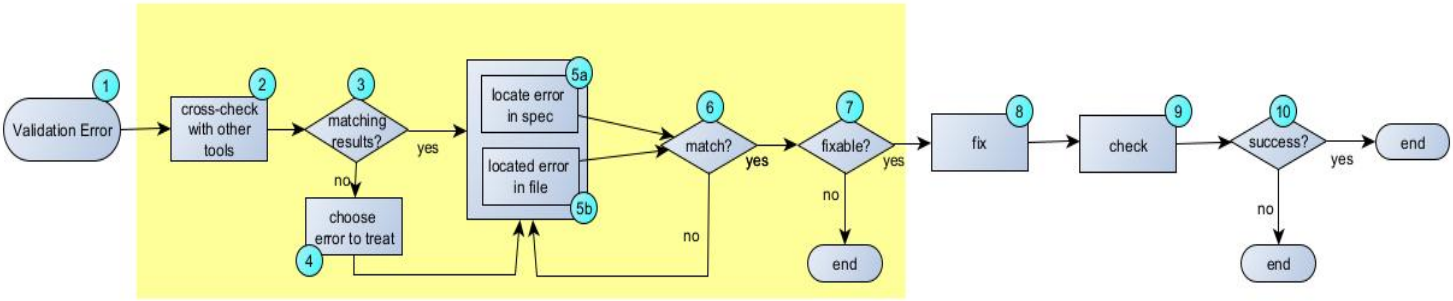


Figure 1: Basic overview of validation error treatment process. The yellow box includes the main analysis steps.

(2) Following a fixed path instead of ad-hoc processes will make it easier to identify gaps in the tools we use.

(3) An easy-to-compare documentation of what we do as a community is the prerequisite to questioning/checking/adapting our processes – it is the first step to next-level digital preservation.

But is a formalized description of what we do when validation fails even possible? This paper shall address exactly that question. After drafting a basic workflow of typical post-validation-error steps, this workflow is tested using two different file formats (TIFF and PDF) with two validation error examples each. In a second step, the basic workflow graphic is adapted according to the analysis outcome and the workflows usability is briefly discussed.

II. Related Work

Digital preservation practice rates "file format validation" as a key task of the ingest process. But what if file format validation fails? While the past decade has put forth new validators such as new JHOVE modules [5], veraPDF [6], DPFManager [7], MediaConch [8] or pdfcpu [9], little progress has been made in describing what to do when things go wrong. With few exceptions, that information largely stays among file format practitioners in our domain [10], [11]. Instead of promoting a broad discussion on these error messages within the community and aiming for joined solution approaches when it comes to handling invalid files, we often find ourselves questioning the process per-se [12].

While Gattuso and Goethals reported on a workflow used to assess and mitigate JHOVE validation errors at the National Library of New Zealand in 2017 [10], little work has been undertaken on formalizing a generic workflow for post-validation-error situations. Even the "Community Owned Workflows (COW)" section of the COPTR Wiki [13] includes only one validation-centric description, which does not really touch on error handling.

III. Methodology

For notation of the process, a simple flowchart style is used in order to make the diagrams easy to understand, thus allowing them to be of benefit to the widest audience possible. In a first step, a basic overview of the process is drafted. The single steps outlined are based on shared community experiences made in the past 10 years of digital preservation practice [1],[2], [10],[11],[15]. Figure 1 shows this basic overview.

The starting point of the workflow is a validation error message, while its ending point as indicated in Figure 1 is the result of the validation error treatment process. In the wider digital preservation process this might be a decision to accept or decline the file. However, capturing this decision is considered outside of the scope of this paper. The process is broken down into two larger categories – the analysis chain (see yellow box in Figure 1) and the "treatment" chain following the analysis. The steps are described in further detail in subsection III A "Definitions".

The basic overview shall be a starting point for documenting the post-validation process. In a next step, the workflow is tested against real-life use cases to see where it works and where it does not work. Of particular interest is the question of how the basic

workflow works when it comes to different file formats and different "severity levels" of validation errors. Since file format characteristics differ widely, two different file format families are chosen as examples to check the workflow against. Both file formats are widely adopted, have an openly available specification and more than one validation tool available. TIFF shall represent file formats that are of comparatively strict and simple structure; PDF shall represent file formats with a comparatively flexible and complex structure. For both file formats two different error messages are chosen – one "fixable" and one "not fixable" error each - to test the workflow description against. While within the scope of this paper all workflow descriptions start with a JHOVE error message, the workflow diagram is kept generic enough to work with any validation tool's error output.

A. Definitions

Before looking at the workflow diagram in further detail, a shared understanding of "validation" needs to be reached. The Community Owned digital Preservation Tool Registry (COPTR) classifies the function validation as a subset of the lifecycle stage ingest, describing it as "(...) the validation of digital files, typically against a file format specification" [32]. The dpc handbook has an even broader approach, stating that file format validation compares an instance of a file format to its expected behaviours [33]. More granular discussions of file format validation [2], [12], [15], [34] differentiate between different error levels of validation, such as "well-formed" and "valid" or "error" and "warning". Within the scope of this paper, (file format) validation is understood as any tool-based method to check a file format instance against a publically available description of the file format's syntax and semantics. This description can be in form of a full standard document, a format specification or a rule set, including a rule set of the validator itself.

The rest of this subsection gives a short overview of each of the workflow steps described in Figure 1 including their necessity and dependency. Necessity of a step depends on pathways chosen – e.g., step 4 ("choose error to treat") is optional, as it depends on more errors than one being present in the validation results.

Step 1: Validation Error (Mandatory)

Description: Starting point of the workflow; error can be from any tool used to validate the syntax and semantics of the file format

Prerequisite: Validation error message; access to the validation tool used; access to the file being validated

Step 2: Cross-check with other Tools (Optional)

Description: If other tools are available to check the validity of the file, these are run to cross-check and potentially gather further information; step is optional since further tools may not be available for all cases

Prerequisite: Availability of further tools to check validity of file

Step 3: Matching Results? (Optional)

Description: If different tool(s) are used to cross-check (step 2), tool outputs are compared to initial validation error message (step 1); the decision whether results match is not necessarily a straightforward task as terminologies may differ between tools

Prerequisite: Cross-check with other tools completed and results documented (step 2)

Step 4: Choose Error to Treat (Optional)

Description: If additional errors were found, a decision needs to be made which error is handled; in some cases errors may be connected to each other, leading to more than one error being handled in the following analysis and fix steps

Prerequisite: Additional tool(s) available (step 2) and additional validation errors found (step 3)

Step 5a: Locate Error in Spec (Mandatory)

Description: Validation tools check against a rule set which is derived from a standard or specification document for the file format; checking the validity of an error requires a comparison of the specification that is being checked against and the position in the file that triggered the error;

Prerequisite: Knowledge of and access to the documentation which the tool checks against (i.e., standards document, specification, schema)

Step 5b: Locate Error in File (Mandatory)

Description: The position in the file that triggered the error is typically referenced in the error message

(e.g., via offset, tag name, chunk, etc); it can be accessed via tools like a hexeditor (for binary formats), an editor (for text based formats) or of a structure parsing tool such as itext RUPS [31] for PDF

Prerequisite: Information about section of file that triggered the error; access to an analysis tool like hexeditor or editor; knowledge of how to navigate through the file formats structure

Step 6: Match? (Mandatory)

Description: Rationale for the error message are compared by checking the rule against the respective section of the file – this allows to check for false positives (validation tool errors); this step also forms the basis for understanding the impact of the tool, resulting in necessary information for a potential fix

Prerequisite: Rule that triggered the error and corresponding section in the file

Step 7: Fixable? (Mandatory)

Description: While some validation errors cannot be fixed, others can, but institutions may elect not to do so, e.g., because the error has no impact on rendering behavior; since the decision not to fix a file leads to the first end marker, step 7 is the last mandatory step in the workflow description

Prerequisite: Understanding of the error message and its impacts; tools / methods to conduct fix

Step 8: Fix (Optional)

Description: Repairing the file within the context of the validation error message (step 1); while some institutions may decide to discard the original after a successful fix, both versions (original and fixed) should be kept until the end of the workflow described here

Prerequisite: Knowledge of a method and availability of tools needed to fix the validation error within the file

Step 9: Check (Optional)

Description: Fixed files are cross-checked by rerunning the tools that produced the original validation error (step 1) as well as, if available, other validation tools (step 2); outcome of check determines whether workflow may need to start over again with a new validation error; in addition to validation checks, content-based integrity checks (where available) may be conducted to verify that actual content of the digital object was unchanged

Prerequisite: Original and repaired file for potential cross-checks; content-based integrity check tools or methods (where available)

Step 10: Success? (Optional)

Description: fixes can be successful or not – the two different outcomes typically serve as hooks for follow-up workflows within an archive (e.g., decline unfixable file)

Prerequisite: Understanding of impact of fix on digital object

IV. Analysis - TIFF

The following section describes processes for two different TIFF validation errors by using the basic flowchart description. The first error is one that can be fixed while the second error is one without a known remedy.

The starting seed validation error always stems from JHOVE v1.26 TIFF-hul 1.9.3[5]. Cross-checking is always completed with DPF Manager v3.5.1 [7] in full-check against Default mode as well as with ExifTool v12.44 [16]. The steps outlined in Figure 1 will be referenced by their respective numbers.

A. TIFF Use Case 1: TIFF-HUL-2 Tag 270 out of sequence

The error and handling described here is similar to that of a previously published blog-post [17]. The error has been reproduced in a file made available as *TIFF_Case-1.tif* in the dataset associated with the paper [18].

Step 1: Validation Error

The JHOVE validation error is *"TIFF-HUL-2: Tag 270 out of sequence."* As additional information, JHOVE gives the offset at which the error occurs: 178

Step 2: Cross-check with other tools

The error is cross-checked with DPF Manager and Exiftool. DPF Manager reports two errors and one warning: *IFD-0007 "Tags must be in strict ascending order" for IFD1* and *IDFE-0002 "Only 7-bit ASCII codes are accepted" for tag 270 ImageDescription"*. In addition, DPFManager lists one warning. However, DPFManager warnings are considered out of scope for this paper as the tool clearly differentiates between errors and warnings. The DPFManager output includes a reference to the part of the TIFF specification that is violated by the file – for both

errors that is "*TIFF Baseline 6: Section 2: TIFF Structure. Page 15*". Exiftool (called with *-validate –warning –a* flags) returns two warnings: *"Entries in IFD0 are out of order"* and "*Tag ID 0x010e ImageDescription out of sequence in IFD0"*.

Step 3: Matching Results?

JHOVE'S *TIFF-HUL-2 "Tag 270 out of sequence"*, DPF Manager's *IFD-0007 "Tags must be in strict ascending order"* and Exiftool's *"Entries in IFD0 are out of order"* and *"Tag ID 0x010e ImageDescription out of sequence in IFD0"* appear to be matching results – although the different referencing of the IFD as IFD0 and IFD1 between DPF Manager and ExifTool are confusing. DPF Manager finds one additional error pertaining to a non-7 bit ASCII Code (in this case, a German Umlaut "ö") in tag 270.

Since the tool set we ran puts forth two different errors, we move along the "no" branch to Step 4.

Step 4: Choose Error to treat

We choose to neglect the DPF Manager Tag 270 non-7-bit-ASCII Character error for now and focus on the original JHOVE error, which was confirmed by DPF Manager and ExifTool.

Step 5A: Locate error in spec

Thanks to the detailed information returned by DPF Manager, we know exactly where to consult the TIFF specification. DPF Manager paraphrases the specification text for us, so we do not necessarily have to go look it up ourselves: *"The entries in an Image File Directory(IFD) must be in strictly ascending order by tag although the values which directory entry points need not be in any particular order"* [19],[20].

Step 5B: Locate error in file

JHOVE navigates us to two locations: while the offset is of little help here as the information in the binary cannot be understood easily, the tag number given in the error message itself is indeed helpful. With a tag viewer like ExifTool, we can extract the tags as they appear in sequence in the file and we can indeed see that tag number 270 is located between 305 and 317, so clearly not in ascending order.

Step 6: Match?

The error message "Tag 270 out of sequence" matches with what we have found in the file and can be verified. We therefore move along the "yes" branch to step 7.

Step 7 - 8: Fixable? & Fix

As described in [17], the error can be fixed with Exifool using the *–P –ImageDescription= -tagsfromfile @ -ImageDescription* flags. We move along the "yes" branch in step 7 and fix the file in step 8. This results in the creation of a new file with the correct tag order while maintaining all timestamp information. The fixed file is included as *TIFF_Case-1_fixed_1.tif* in the dataset associated with this paper [18].

Step 9 – End: Check & Success?

The success of the fix can be verified by re-running the file through JHOVE, DPF Manager and ExifTool as well as by manually re-inspecting the file as described in step 5B. In addition, the integrity of the image data can be verified by comparing the hash of the image data in the old file to that of the new file. This can be achieved with ImageMagick [30] using *identify –quiet –format "%#"*. We conclude the handling of this instance of TIFF-HUL 2 Tag 270 out of sequence by moving along the "Yes" branch to the workflow's "End" marker.

The case-specific workflow diagram is included as Appendix A1 to this paper.

While the workflow has been completed successfully for the specific JHOVE error used as a starting seed, we did encounter additional errors along the way. When checking the fixed file in Step 9, JHOVE returned the object as well-formed and valid, whereas DPF Manager continued to report the IDFE-0002 Error "Only 7-bit ASCII codes are accepted" for tag 270. While the error can be easily treated using the same workflow methodology, it imposes questions on how multiple error treatment should be reflected in the description. This question is elaborated on further in the Discussion section of this paper.

B. TIFF Case 2: TIFF-HUL-28 StripOffsets inconsistent with StripByteCounts

The error and handling described here is similar to that of a previously published blog-post [21]. The error has been reproduced in the file *TIFF_Case-2.tif* that is available in the dataset associated with this paper [18].

Step 1: Validation Error

The JHOVE validation error is "*TIFF-HUL-28: StripOffsets inconsistent with StripByteCounts: 1 != 55*". No further information is given.

Step 2: Cross-check with other tools

The error is cross-checked with DPF Manager and Exiftool. DPF Manager reports two errors: *IDFE-0002 "Only 7-bit ASCII codes are accepted" for tag 270 ImageDescription* as well as *STRIPS-0005 "Inconsistent strip lengths, the cardinality of stripoffsets and StripsBytesCount must match"*. In addition, DPFManager lists one warning, which has no impact on the file and shall be neglected in the scope of this paper. The DPF Manager output includes a reference to the part of the file format specification that is violated by the file – for *STRIPS-0005* that is *"TIFF Baseline 6: Section 8: Baseline Field Reference Guide, Page 40*". ExifTool (called with *-validate –warning –a* flags) returns one warning: *"Wrong number of values in IFD0 0x0111 StripOffsets".*

Step 3: Matching Results?

JHOVE's *TIFF-HUL 28 "StripOffsets inconsistent with StripByteCounts: 1 != 55"*, DPF Manager's *STRIPS-0005 "Inconsistent strip lengths, the cardinality of stripoffsets and StripBytesCount must match"* and ExifTool's "*Wrong number of values in IFD0 0x0111 StripOffsets"* appear to be matching results. DPF Manager finds one additional error pertaining to a non-7 bit ASCII-Code (in this case, a German "ß") in Tag 270.

Since the tool set we ran puts forth two different errors, we move along the "no" branch to Step 4.

Step 4: Choose Error to treat

We choose to neglect the Tag 270 non-7 bit ASCII character error and focus on the original JHOVE error.

Step 5A: Locate error in spec

Again, DPF Manager paraphrases the section of the specification: "*The cardinality of stripOffsets and the cardinality of StripsBytesCounts must be the same"*.

Step 5B: Locate error in file

We can locate the error by extracting the values from the binary data of the respective tags. This is possible by using ExifTool's *–b* option. Comparing the values in question, we see for *TIFF_Case-2.tif* that StripOffsets contains 1 value, StripByteCounts contains 55 values.

Step 6: Match?

The error message *"StripOffsets inconsistent with StripbyteCounts: 1 != 55"* can be verified in the file. We therefore move along the "yes" branch to step 7.

Step 7 - End: Fixable?

Unfortunately, the error means that the image data cannot be extracted from the file correctly [21]. Since the error is not fixable, we move along the "no" branch of step 7 and reach the end of the workflow here.

The case-specific workflow diagram is included as Appendix A2 to this paper.

While the workflow could be applied correctly to the use case, we discovered that we need to rely on outside information or knowledge when it comes to the question of fixable errors. This topic will be picked up later in the Discussion section of this paper.

V. Analysis – PDF

The following section describes processes for two different PDF validation errors against the basic flowchart description. The first error is one that can be easily fixed, while the second error is one without a known remedy.

The starting seed validation error always stems from JHOVE v1.26 PDF-hul 1.12.3 [5]. Cross-checking is always completed with pdfcpu (v.0.4.0. dev, *validation –mode strict*) [9] and qpdf (v. 9.1.1, *--check –verbose* options) [22]. The steps outlined in Figure 1 are referenced by their respective numbers.

A. PDF Use Case 1: PDF-HUL-137 No Pdf Header

The error of "junk data" before the header is common, especially in some older PDF files [23]. For this use case, an example of such a case discovered "in the wild" is used. The file is made available as *PDF_Case-1.pdf* via the dataset associated with this paper [18].

Step 1: Validation Error

The JHOVE validation error is *"PDF-HUL-137: No PDF Header"*. The Offset is given as 0.

Step 2: Cross-check with other tools

The error is cross-checked with pdfcpu and qpdf. Pdfcpu returns the validation error *"xRefTable failed: pdfcpu: headerVersion: corrupt pdf stream – no header version available"*, whereas qpdf returns no error.

Step 3: Matching results?

JHOVE's *PDF-HUL 137* matches the *"no header version available"* message thrown by pdfcpu. Since there is only one error to treat, we move along the "yes" branch directly to step 5A.

Step 4: Choose error to treat

Step 4 is skipped as we moved directly to step 5A from the "yes" branch in step 3.

Step 5A: Locate error in spec

ISO 32000-1:2018 states in section 7.5.2 that *"The first line of a PDF file shall be a header consisting of the 5 characters %PDF- followed by a version number of the form 1.N where N is a digit between 0 and 7"* [24].

Step 5B: Locate error in file

Viewing the file in a hex editor, we can easily see that the first line is not the version info. There are 128 additional bytes before the %PDF-1.3 declaration.

Step 6: Match?

The error message *"No PDF Header"* can be verified via file inspection. We therefore move along the "yes" branch to step 7.

Step 7 – 8: Fixable? & Fix

The PDF can be fixed by removing the 128 bytes before the %PDF-1.3 declaration, as they are deemed "junk data" [25]. We move along the "yes" branch in step 7 and fix the error in step 8.

Step 9 – End: Check & Success?

The fix can be verified by rerunning the fixed file through JHOVE and pdfcpu. The file is now returned as well-formed and valid and we move along the "yes" branch in step 10 to the end of the workflow.

The fixed file is made available as *PDF_Case-1_fixed.pdf* in the dataset associated with this paper [18].

In addition, Adobe Acrobat Professional offers a compare tool, via which two PDFs can be compared and a difference report be generated. The case-specific workflow diagram is included as Appendix A3 to this paper.

B. PDF Use Case 2: PDF-HUL-38 Invalid Object Definition

The last use case is a difficult and unsolved PDF case. The issue has been previously discussed in a blog post [1]. It is chosen to test how well the proposed workflow diagram is applicable to complicated cases where it is hard to pinpoint the error. An example of such a case discovered "in the wild" is used and made available as *PDF_Case-2.pdf* via the dataset associated with this paper [18].

Step 1: Validation Error

The JHOVE validation error is *"PDF-HUL-38: Invalid Object Definition".* The Offset is given as 285259. For this particular test file, JHOVE throws another error as well: *PDF-HUL 87 "File header gives version as 1.3, but catalog dictionary gives version as 1.4".* Since the workflow description takes exactly 1 validation message as a starting point, and since PDF-HUL-87 is an error that can be neglected [26] we will focus on PDF-HUL-38.

Step 2: Cross-check with other tools

The error is cross-checked with pdfcpu and qpdf. Pdfcpu returns the validation error *"dereferenceObject: problem dereferencing object 91: pdfcpu: ParseObjectAttributes: can't find 'obj'"*. Qpdf shows a total of 28 error messages. 19 of those are *"object has offset 0"* for different objectIds, 5 of those comment on missing or incorrect entries in different objectIds, 2 deal with object 91 (*"expected n n obj"* and *"0 not found in file after regenerating cross-reference table"* and 2 of the error messages pertain to the document as a whole (*"file is damaged"* and *"attempting to reconstruct cross-reference table"*).

Step 3: Matching results?

It is hard to figure out whether the different error messages actually describe the same problem. Both, pdfcpu and qpdf have an error pointing to obj 91. Unfortunately, the JHOVE offset does not lead to obj 91 but to obj 194. None of the other tools have an error message pointing to obj 194. Qpdf is the only tool that reports the file as damaged. Since the error messages point to different sections of the file, we move along the "no" branch to step 4 to choose which error to treat and analyze in the following steps.

Step 4: Choose error to treat

Even though none of the other errors match with the JHOVE error message, we will stick with our starting seed message and attempt to treat the PDF-HUL-87 for obj 194 first.

Step 5A: Locate error in spec

According to the JHOVE error message documentation the error message occurs when a keyword other than “obj” was found while paring an indirect object definition [27]. According to section 7.3.10 of the specification Object definitions need to follow the form “<obj.number> <obj.generation> obj” [24].

Step 5B: Locate error in file

We already navigated to the respective obj via the JHOVE offset in step 3 to cross-check the result against that of other validation tools. Indeed instead of the expected “194 0 obj” we find a “194 00obk”.

Step 6: Match?

The error message *“Invalid Object Definition”* can be verified via file inspection and we move along the “yes” branch to step 7.

Step 7-10: Fixable? – Success?

We can replace the faulty “194 00obk” in the HexEditor with “194 0 obj”. However, when checking the file with the same tools used in Step 2, qpdf and pdfcpu still show the same error messages as before whereas JHOVE now shows a different error message – *“PDF-HUL-66 Lexical Error”*. Since the file is mangled from a specific point onwards, the post-validation workflow process could be repeated countless times without reaching a successful result. The process is aborted here.

The case specific workflow diagram is included as Appendix A4 to this paper. The PDF with the applied fix for PDF-HUL-38 as per Step 8 is included in the dataset as *PDF_Case-2_fixed1.pdf* [18].

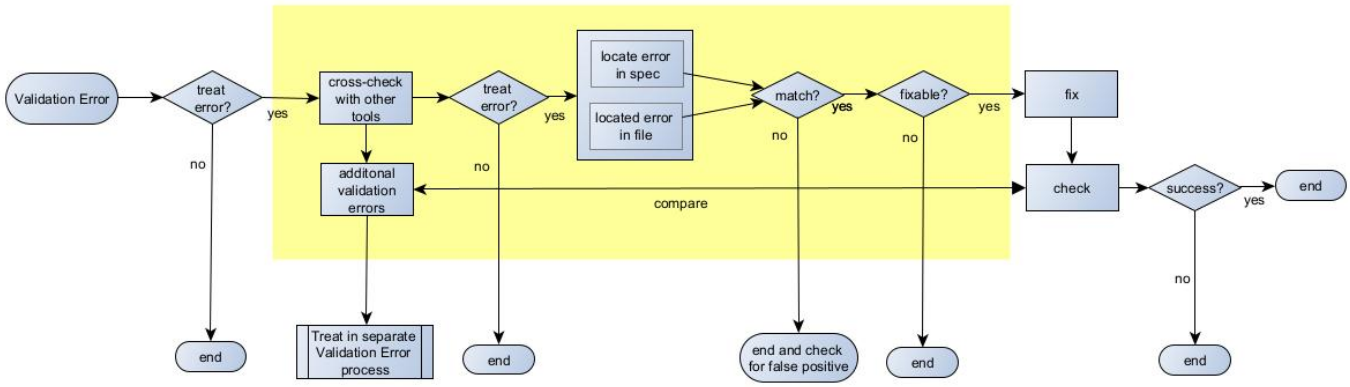


Figure 2: Revised basic overview of validation error treatment process. The yellow box includes the main analysis steps.

While the workflow described how the specific JHOVE error PDF-HUL-38 was treated, the digital object itself seems to have a bigger problem, which all reported errors are connected to. The question on whether this interdependency can be modeled in the workflow will be touched upon in the discussion section.

VI. Discussion

In testing the workflow diagram against the use cases we discovered several issues that should be reflected in an updated diagram version. This updated version is presented as Figure 2. A common strand was that for all use cases external information needed to be consulted to address the issues. This included error message description for JHOVE in the GitHub Wiki, blog posts or uncodified information known through practice. This is where tools can be easily improved – e.g., the JHOVE GUI could include pointers to the specification the same way that DPFManager does. This would especially help less experienced users in making the connection between the error and the expected file format syntax. As already discussed within the community, different error levels such as warnings in addition to errors would be a beneficial addition to JHOVE as well [28].

Both, the TIFF and PDF use cases included warnings or error messages which we did not treat (e.g., PDF-HUL-87 in the second PDF use case). While an in-depth discussion of these errors is out of scope for this paper, the question of how to model the decision not to treat an error is relevant to the diagram. But where in the workflow does that decision take place?

While the first draft of the diagram presented in Figure 1 works on a “blank page” assumption, i.e., no

knowledge of validators and error messages exist, the decision to ignore an error is always based on existing knowledge. Often, errors such as the aforementioned PDF-HUL-87 or the "warning" messages included in the ExifTool output are specific to a tool. While a "warning" as opposed to an error could imply that no direct action is necessary, the actual decision not to treat the issue is always an individual one that may depend on an institutional policy. This binds the decision directly to a business rule, but also to the error message and its producing tool itself - the option not to treat the error therefore needs to be located at the beginning of the workflow. In some cases, however, the decision may depend on a "second opinion", i.e. a verification by a tool used to cross-check. Therefore, an optional additional "treat error" decision should be available after the cross-checking step. Figure 2 shows the updated diagram, with two "treat error" steps added directly after the validation error starting seed and again after the step "cross-check with other tools".

Knowing which errors not to treat goes hand in hand with knowing what errors should be treated. Once either an understanding of the error message itself and its correlation to the specification and file's actual syntax, or a solid trust in the validation tool has been established, the steps "locate error in spec", "locate error in file" and "match" may become unnecessary. These steps should therefore be optional. However, the same argument can be made for "cross-check with other tools", as this may become no longer needed once a high level of trust in one tool's ability is gained. Since the necessity of each step is therefore subjective, depending on the knowledge of those following the workflow, the mandatory / optional descriptors are removed from the updated workflow description.

But what if we come across multiple error messages within a file? And is the validation error message really the correct starting seed, or should the starting point instead be the digital object? In the first TIFF use case, JHOVE had only reported one error, whereas DPF Manager put forth a second error to be fixed. As shown in the use cases, validation error handling can become a complex task. In addition, we have to differentiate between error messages that are dependent on each other and those that are not. . The *"non-7-bit-ASCII"* error message introduced in TIFF Use case 1 is clearly an independent error message, whereas the 28 qpdf error messages found in the second PDF use case appear to depend on each other or on the same root cause. However, we want the workflow diagram to be an easy communication tool. Trying to model more than one validation error message at once in a diagram would make the diagram overly complex. Therefore, the decision is made to outsource additional error handling into new "Validation Error Handling" processes. The option to do so is added as a step resulting from the "cross-check with other tools". Since it might be helpful to understand if errors are dependent on each other or not, an optional "compare" connection was added to the diagram between the "additional validation errors" resulting from the initial cross-check and potential output from the "check" step post fixing.

Another issue exists with the conditional based on the match between the format specification and the error in file. In the first version of the diagram as presented in Figure 1, a successful matching leads to a fix, whereas an unsuccessful match leads to a re-evaluation of the specification and the error message. This could easily create endless loop if the connection simply does not exist, e.g. in case of a false positive returned by the validation tool. Instead of looping back to the comparison, a "no match" should exit the process and result in an evaluation of the validation error as a potential false positive.

Figure 2 presents the updated diagram with all changes included. The numbering and necessity of the steps has been removed due to aforementioned reasons.

VII. Conclusion and Outlook

The paper presented a first draft of a formalized methodology in form of a basic overview diagram for post-validation-error process steps. This first version was tested against four real-life use cases and updated based on the findings. As a general observation, the diagram outlines common steps but does not, of course, contain all the answers for what to do. However, having a structured documentation instead of having to sift through blog posts, wikis etc. to find the answer might make it easier for people to learn from and build on experiences made by others.

The introduction section listed three potential key benefits of such a formalized overview – one of those already proved achievable in form of recommendations for tool improvements made in

the discussion section of this paper. Whether the diagram can be a vehicle for more effective communication between practitioners and those just starting out and whether it can aid us in improving our processes remains to be seen.

A next step for this work is to collect community feedback and model more use cases on the updated version of the diagram. These use cases will then be included in the COPTR COW section. A long-term goal could also be to model validation error decisions in the Preservation Action Registry PAR [29].

1. REFERENCES

[1] Lindlar, M. "Trouble-shooting PDF validation errors – a case of PDF-HUL-38". Blogpost at the Open Preservation Foundation, Published on 27. November 2022. https://openpreservation.org/blogs/trouble-shooting-pdf-validation-errors-a-case-of-pdf-hul-38/

[2] Lehtonen, J. et al. "PDF Mayhem: Is Broken Really Broken?" In: Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, Massachusetts, USA, Sept. 24-28 2018. https://doi.org/10.17605/OSF.IO/FZXC9

[3] Digital Preservation Coalition. "iPRES 2022 – Bake Off: Full Menu – All You Can Eat". Recording of the Digital Preservation Bake Off Session held at the 18th International Conference on Digital Preservation, iPRES 2022, Glasgow, Scotland, Set. 12 – 16 2022. https://youtu.be/fj4Qg_Kj-xc

[4] OPF ICRFF Working Group. "International Comparison of Recommended File Formats" Version 1.2, April 2022. https://openpreservation.org/resources/member-groups/international-comparison-of-recommended-file-formats/

[5] JHOVE https://jhove.openpreservation.org/

[6] veraPDF https://verapdf.org/

[7] DPF Manager http://dpfmanager.org/

[8] Mediaconch https://mediaarea.net/MediaConch

[9] Pdfcpu https://pdfcpu.io/

[10] Gattuso, J., Goethals, A. "The Tip of the Validation Iceberg – Addressing JHOVE-based file validation warnings" In: Proceedings of the 14th International Conference on Digital Preservation, Kyoto, Japan, 25 – 29 September 2017. https://hdl.handle.net/11353/10.1424902

[11] Töwe, M., Geisser, F., Suri, R. "To Act or Not to Act – Handling File Format Identification Issues in Practice". In: Proceedings of the 13th International Conference on Digital Preservation, Bern, Switzerland, October 3 – 6, 2016. https://hdl.handle.net/11353/10.503183

[12] Whatley, P. "A valdediction for validation?" Blogpost at the DigitalPreservationCoalition. Published on 11 October 2018. https://www.dpconline.org/blog/a-valediction-for-validation

[13] COPTR Wiki: Community Owned Workflows (COW). https://coptr.digipres.org/index.php/Workflow:Community_Owned_Workflows

[14] Van der Knijff, J. "PDF processing and analysis with open –source tools". Blogpost at bitsgalore. Published on 06 September 2021. https://www.bitsgalore.org/2021/09/06/pdf-processing-and-analysis-with-open-source-tools

[15] Lindlar, M. Tunnat, Y. "How Valid is your Validation? A Closer Look behind the Curtain of JHOVE". In: International Journal of Digital Curation. Vol. 12, No. 2 (2017). https://doi.org/10.2218/ijdc.v12i2.578

[16] ExifTool. https://exiftool.org/

[17] Lindlar, M. "Troubles with TIFF: Tag 270 out of sequence". Blogpost at the Open Preservation Foundation. Published on 19 March 2020. https://openpreservation.org/blogs/troubles-with-tiff-tag-270-out-of-sequence/

[18] Lindlar, M. Dataset for this paper – currently at https://drive.google.com/drive/folders/1WmY0-nWvjq5aKwBdkvxu43_7h7iiLeU4?usp=sharing will be moved to Zenodo upon acceptance of paper

[19] DPS Manager. "Reference Documentation". http://dpfmanager.org/reference-documentation.html

[20] Aldus Developers Desk. "TIFF. Revision 6.0. Final" June 3, 1992.

[21] Lindlar, M. "Troubles with TIFF: StripOffsets inconsistent with StripByte Counts". Blogpost at the Open Preservation Foundation. Published on 12 April 2020. https://openpreservation.org/blogs/troubles-with-tiff-stripoffsets-inconsistent-with-stripbytecounts

[22] Qpdf https://qpdf.readthedocs.io/

[23] Lindlar, M., Tunnat, Y. "Time-travel with PRONOM: the 4th dimension of DROID". In: Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, Massachusetts, USA, Sept. 24-28 2018. https://doi.org/10.5281/zenodo.3517767

[24] ISO/TC 171/SC 2. "ISO 32000-1:2008 Document management – Portable document format – Part 1: PDF 1.7". 2008.

[25] OPF. "PDF-HUL-137". Error Message Wiki on github. https://github.com/openpreserve/jhove/wiki/PDF-hul-Messages-2#pdf-hul-137

[26] OPF. "PDF-HUL-87". Error Message Wiki on github. https://github.com/openpreserve/jhove/wiki/PDF-hul-Messages-2#pdf-hul-87

[27] OPF. "PDF-HUL-38". Error Message Wiki on github. https://github.com/openpreserve/jhove/wiki/PDF-hul-Messages#pdf-hul-38

[28] OPF. "Does JHOVE need a warning Message type?" JHOVE github issue #638 Discussion. https://github.com/openpreserve/jhove/issues/638

[29] Preservation Action Registries (PAR). https://parcore.org/

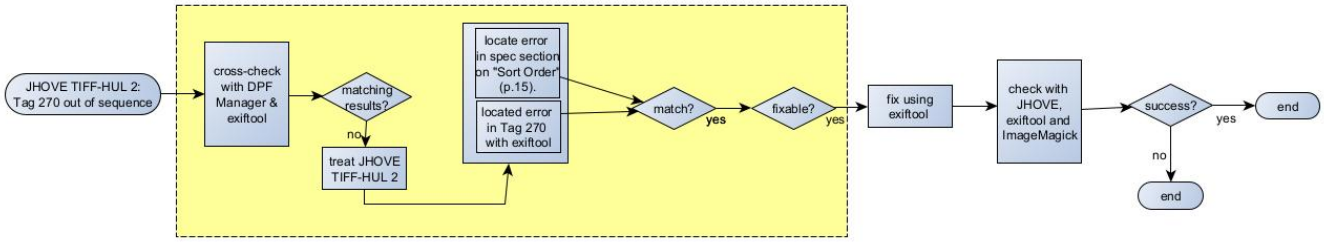
[30] ImageMagick. https://imagemagick.org/

[31] iText RUPS: https://github.com/itext/i7j-rups

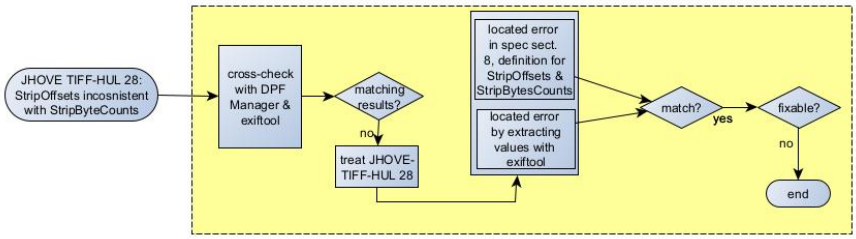
[32] COPTR function category"Validation": https://coptr.digipres.org/index.php/Validation

[33] dpc. "File formats and standards" In: Digital Preservation Handbook, 2nd Edition. 2015 https://www.dpconline.org/handbook .

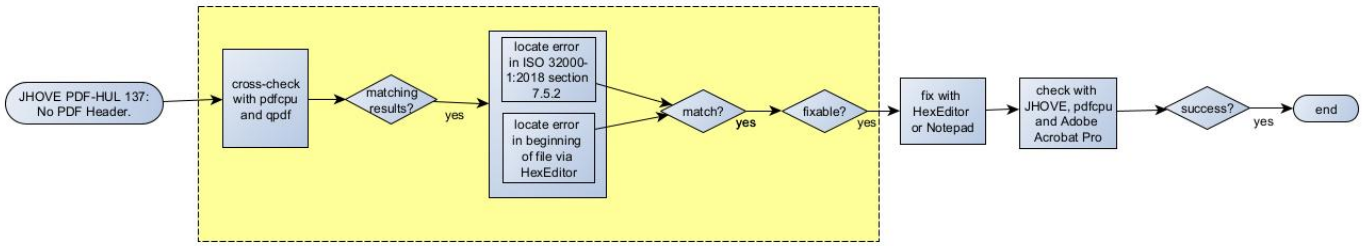
Appendix A1: Workflow for TIFF Use Case 1 – TIFF-HUL-2 Tag out of Sequence



Appendix A2: Workflow for TIFF Use Case 2 – TIFF-HUL-28: StripOffsets inconsistent with StripByteCounts



Appendix A3: Workflow for PDF Use Case 1 – PDF-HUL 137: No PDF Header



Appendix A4: Workflow for PDF Use Case 2 – PDF-HUL-38: Invalid Object Definition