A PDF Test-Set for Well-Formedness Validation in JHOVE - The Good, the Bad and the Ugly

Michelle Lindlar
TIB - Leibniz Information Centre of Science and Technology
Welfengarten 1B
Hannover, Germany 30167
michelle.lindlar@tib.eu

Yvonne Tunnat
ZBW - Leibniz Inforation Centre of Economics
Düsternbrooker Weg 120
Kiel, Germany 24105
y.tunnat@zbw.eu

Carl Wilson
OPF - Open Preservation Foundation
c/o The British Library
Boston Spa, United Kingdom LS23 7BQ
carl@openpreservation.org

ABSTRACT

Digital preservation and active software stewardship are both cyclical processes. While digital preservation strategies have to be reevaluated regularly to ensure that they still meet technological and organizational requirements, software needs to be tested with every new release to ensure that it functions correctly. JHOVE is an open source format validation tool which plays a central role in many digital preservation workflows and the PDF module is one of its most important features. Unlike tools such as Adobe PreFlight or veraPDF which check against requirements at profile level, JHOVE's PDF-module is the only tool that can validate the syntax and structure of PDF files. Despite JHOVE's widespread and long-standing adoption, the underlying validation rules are not formally or thoroughly tested, leading to bugs going undetected for a long time. Furthermore, there is no ground-truth data set which can be used to understand and test PDF validation at the structural level. The authors present a corpus of light-weight files designed to test the validation criteria of JHOVE's PDF module against "well-formedness". We conclude by measuring the code coverage of the test corpus within JHOVE PDF validation and by feeding detected inconsistencies of the PDF-module back into the open source development process.

KEYWORDS

file format validation, PDF, test data, quality assurance, JHOVE

ACM Reference format:
Michelle Lindlar, Yvonne Tunnat, and Carl Wilson. 2017. A PDF Test-Set for Well-Formedness Validation in JHOVE - The Good, the Bad and the Ugly. In *Proceedings of iPRES Conference, Kyoto, Japan, September 2017 (iPRES 2017)*, 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

File format validation is a central task in digital preservation processes, giving insight into the degree with which the digital object complies with the specification of the file format it purports to be. For complex formats such as PDF, which allow for a multitude of content types and variations, such as embedded AV material or embedded and non-embedded fonts, validation poses a challenging problem. While software to validate digital object's against PDF profile requirements such as PDF/A[1] or PDF/X exist, they typically focus on the requirements of the profile and do not take the syntactical and structural requirements of the underlying PDF format into account [8]. As of today, the go-to validator for the PDF format is the open source tool JHOVE [23][2]. The initial development of JHOVE dates back to 2003-2008 and the tool has been widely used by digital archives since.

Digital preservation and active software stewardship are both cyclical processes. While digital preservation strategies have to be regularly reevaluated to ensure they continue to meet technological and organizational requirements, software needs to be tested with every new release to ensure that it functions correctly. Despite JHOVE's widespread and long-standing adoption, the underlying validation rules are not formally or thoroughly tested, leading to bugs which can go undetected for a long time. Formal testing for complex software such as file format validators has to be automated. However, a requirement for such automated testing processes is a ground-truth as a point of reference, ideally manifested in a light-weight test set. This test set can be used to check the validator's capability to enforce specific clauses in the format specification. In the case of PDF validation in general and JHOVE specifically, no such test-set has been available until now.

This paper describes the authors' efforts to narrow this gap by building a light-weight test-set for PDF validation. The test set focuses on the validation against structural and syntactical requirements[3] of the PDF file format as described in the ISO 32000-1:2008 standard for PDF 1.7. It will not look at particular profile requirements such as those described in the ISO 19005 series for PDF/A. As the standard does not make a clear distinction between well-formed and valid requirements, these are derived by looking at required structural parts of any PDF object, namely: a header, a body consisting of a minimal set of objects, a cross-reference table and a trailer (see Figure 1). While JHOVE only supports PDF features up to version 1.7, the cases implemented in the test set are common to all PDF versions. The aim of test set is threefold:

(1) to establish a ground truth for what is not well-formed
(2) to test the JHOVE software against that ground truth
(3) to improve automated regression testing

[1]For PDF/A, e.g.: veraPDF, Callas pdfapilot, PDFTron, 3-Heightsffl
[2]While the JHOVE framework includes a variety of validation modules, this paper limits the scope to the PDF-module. Within the context of this paper JHOVE is therefore used as a synonym for the JHOVE framework's PDF-module.
[3]Syntactical and structural requirements equal JHOVE's well-formedness criteria. Please refer to section 2.1 for further discussion.

Section 2 of this paper will introduce the concept of file format validation and give insight into the development of JHOVE in general and the PDF module in particular to provide a contextual framework for the test set work. Section 3 will introduce the methodology used for the construction of the test set as well as for measuring and describing the automated regression testing gap. Section 4 describes the test set itself and the results of running the JHOVE PDF-module[4] across the test set. To introduce a second point of reference, each test file is also rendered using a suitable application[5]. While the ability to render a file correctly does not guarantee that it is well-formed, incorrectly displayed content or the failure to render often indicates that the file is not well-formed. Section 5 discusses the impact of the test set as a ground-truth and as an improving factor in current JHOVE code as well as in existing automated regression processes. We conclude with section 6, highlighting possibilities for further work building on the test set described in this paper.

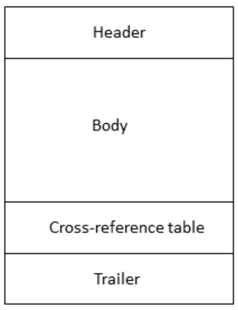

Figure 1: Basic PDF structure

2 BACKGROUND AND RELATED WORK

File format validation is a challenging task. Section 2.1 describes the motivation behind and general approach to file format validation, sections 2.2 and 2.4 illustrate how this challenge was met in the development of the JHOVE framework and the PDF-module, respectively. As aforementioned, very few efforts have been undertaken to validate against standard requirements of PDF at a structural and syntactical level, a recent exception being Caradoc [8]. Written in OCaml and first introduced in 2016, Caradoc is still in its beta stage[6]. As opposed to JHOVE, which skips over unsupported structures thus considering them valid by default, Caradoc sets out to take a whitelist approach, considering unknown features as suspicious by default and flagging them as invalid. While this is a thorough approach, the current implementation of Caradoc only serves as a proof-of-concept, containing rules for a very limited number of PDF features. Due to this, the vast majority of "real-life" PDFs currently fail validation with Caradoc.

[4]At the point of writing the latest available version is JHOVE 1.16 framework, which wraps PDF-module v1.8
[5]The rendering application used is Adobe Acrobat Professional 11.0.15.2
[6]See https://github.com/ANSSI-FR/caradoc for beta 0.3 of the Caradoc tool

Hence, the digital preservation community largely relies on JHOVE for validation - despite known bugs[7]. The adoption of and ongoing work on JHOVE will be introduced in section 2.3, further motivating the relevance of and urgent need for thorough regression testing and ground-truth data.

2.1 File Format Validation

File format validation is the process of checking an object's conformance to syntactic and semantic rules of the format it purports to be. As such, it is closely related to file format identification. While most pattern based identification tools such as DROID or file rely on short signatures such as magic numbers, full identification requires an analysis of the entire bit-stream and a comparison to the structure and semantics prescribed by the file format's specification [1]. To illustrate, consider the following minimal PDF code of the file minimal_test.pdf:

```
%PDF-1.4
%%EOF
```

Minimal_test.pdf is identified as PDF 1.4 by standard file format identification tools[8]. JHOVE, however, recognizes that the object is *Not well-formed*, indicating problems at the basic structural level of the file format level which the object purports to be. Ideally, the normative syntactic and semantic rules used to check the validity of an object are taken from the file format's authoritative specification. However, in many cases a specification may not exist or may not be publicly available. Format specifications not written within an official standardization context present another problem. These are often ambiguous and therefore open to interpretation [2]. Ambiguities in the PDF specifications published by Adobe have lead to a rather broad interpretation of the file formats syntactical and semantic makeup. This, in return, has lead to PDF rendering software being forgiving towards many violations, resulting in files which are strictly speaking invalid but are still renderable and usable [2].

Format validation is usually broken down into two conformance levels - determining whether an object is *well-formed and valid.* The W3C Extensible Markup Language Standard [31], for example, clearly defines the constraints of a well-formed XML object. In short, a well-formed XML document must contain exactly one root element, consist of one or more correctly nested and delimited elements and follow the regulations specified for entities. While well-formed XML objects comply with the XML specification, valid XML objects comply with an XML schema. In short, well-formedness addresses the syntactic correctness while validity describes the semantic correctness of an object's conformity to the file format it purports to be. JHOVE file format modules adhere to this two-tiered conformance checking. The validation rules implemented in the TIFF module, for example, determine a file to be well-formed if the beginning of the file is an 8 byte header followed by a sequence of Image File Directories, which in return are each composed of a 2 byte entry count and a series of 8 byte tagged entries. The module defines an object as being valid if it meets certain additional

[7]E.g., see comment from original JHOVE developer Gary McGath on 2014-07-10: "The PDF module has a history of bugs relating to page trees, (...). If other software doesn't complain, I'd be inclined to call this a JHOVE bug." https://sourceforge.net/p/jhove/discussion/797887/thread/2050dc83/#c70f
[8]Tested with: DROID 6.2.1,Signature File V88, Siegfried 1.5.0 (both identified via signature pattern); TrID/32 v2.24 (findings: 100% PDF without PDF version output) and File for windows v5.03

semantic-level rules, such as that TileWidth (322) and TileLength (323) values are integral multiples of 16 [14]. Most format modules consider well-formedness a prerequisite of validity.[9]

Validation rules for the JHOVE PDF-module will be discussed further in section 2.4.

2.2 Brief History of JHOVE

JHOVE is by no means a new tool to the digital preservation community. The idea of the JSTOR/Harvard Object Validation Environment dates back to 2003 [7]. Partially funded by the Andrew W. Mellon Foundation [24], initial development of the driver and API layers as well as the ASCII, UTF-8, PDF, TIFF, GIF, JPEG and XML modules took 10 months and involved 1.35 full-time equivalents (0.10 project management, 0.25 senior analyst, 1 developer)[7]. With the initial release of version 1.0 in May of 2005, work on JHOVE continued under the auspices of the JSTOR Electronic-Archiving Initiative (now Portico) and the Harvard University Library until 2008. In late 2008, the California Digital Library, Portico and Stanford University secured Library of Congress funding under the National Digital Information Infrastructure Preservation Program (NDIIPP) for a follow-up JHOVE2 project. The project, which ran for two years, was based on the observation that the original JHOVE, even though extensively used, had "revealed a number of limitations imposed by idiosyncrasies of design and implementation" [2]. JHOVE2 was conceptualized to be a complete re-factoring of the software, allowing for simplified integration, containing streamlined APIs and including modules for file formats previously not covered in JHOVE. Two major conceptual changes in the approach to file format characterization were the introduction of a more sophisticated data model. While JHOVE works under the assumption that 1 object = 1 file = 1 format, JHOVE2 allowed for complex objects, shifting the module to 1 object = m files = n objects. A second change was made by decoupling file format identification from validation. While JHOVE conducts file format identification by iteratively calling each existing module until one reports the file to be valid, JHOVE2 relied on DROID for initial file format identification. With the move towards JHOVE2, Harvard University Library's JHOVE developer Gary McGath left, asking for continued custody of the JHOVE code [19] which he facilitated through a SourceForge project. In the following seven years, McGath oversaw the release of 11 versions, including several updates to the PDF-module. In 2013 McGath moved JHOVE to GitHub, which had by then overtaken SourceForge as the code platform of choice [24]. In March of 2014, Gary McGath announced that he could no longer maintain the software by himself [18], leading to the Open Preservation Foundation (OPF) taking over JHOVE stewardship and moving the code to the OPF GitHub repository, where it remains open source under a GNU Lesser General Public License (LGPL). Since then, the OPF has offered software supporters and members the chance to steer maintenance and development activities through the JHOVE product board. With the move to the OPF GitHub repository, the versioning method has changed. Minor version numbers are used for production releases (e.g. 1.16), while odd numbers indicate development releases (e.g. 1.15). Early OPF development efforts focused on providing a more user-friendly installer (version 1.12). Version 1.14 saw the introduction of three new file format modules: WARC, gzip and PNG, and prototype regression testing tools [25], which are discussed in section 3.

[9]While this statement holds true to the original HUL modules as documented on the Sourceforge site [21], information about requirements for well-formedness and validity for the following non-hul modules are unfortunately lacking: ZIP, MP3, WARC, PNG [26] [25]

2.3 JHOVE Adoption

While JHOVE2 never achieved wide adoption in the community [20], [24], and development has been dormant since 2013 [6], JHOVE remains an important tool in many digital preservation workflows. In the 2015 OPF community survey JHOVE was - tied with DROID - ranked by the 132 respondents from around the globe as the most important tool in digital preservation. This is also reflected in the download numbers. Starting with a moderate 30 downloads per week in 2009, the adoption of JHOVE quickly grew reaching 300-400 downloads per week in early 2013 [24]. Since its first release until today JHOVE remains a standard tool mentioned in state-of-the-art system descriptions and best practice reports [15],[32], [27],[9]. Despite the tool's popularity neither exhaustive documentation of the validation rules nor accompanying information to the different error messages exist. Up to today the most comprehensive documentation is still that provided by the original JSTOR/Harvard Project in 2008. To address this gap, the OPF set up the "Document Interest Group" (DIG)[10] in early 2015, which aims to improve JHOVE and the interpretation of the error messages for textual data modules such as the PDF-hul. A first step was the creation of wiki-based documentation of the error messages [11]. In 2016 the OPF DIG conducted the first "JHOVE hack day" [22], leading a large community effort to catalog the error messages for the different JHOVE format modules in a systematic way. Recently, a series of validation tool benchmarks have been conducted, focusing on the identification aspect of JHOVE in juxtaposition to other tools [27] or comparing the validation output of JHOVE's wave [30], TIFF [29], [16], and JPEG [28] modules to the output of other tools that can characterise and validate the respective file format families.

2.4 PDF Module

The PDF-hul module has been frequently updated since JHOVE's 1.0 inception in 2008. With the exception of the 1.6, 1.9 and 1.11 framework releases, every JHOVE release has seen updates to the PDF module, resulting in new versions of the module. Improvements range from handling of parameters in accordance with the specification (version 1.3) to optimizations of the parser and the module's memory use (version 1.10) [25]. Changes made in the PDF-module may change the outcome of validation results for a file. As such, JHOVE's most recent 1.16 version included PDF-hul version 1.8, which fixed two major bugs in the code. These had been present from the start of development, leading to false validation errors relating to invalid page dictionary objects and improperly constructed page trees [17]. While a number of fixes have improved PDF/A validation [25], JHOVE has been proven unsuitable for PDF/A validation [10] [12]. The coverage of PDF versions hasn't changed since PDF-hul 1.0; for "plain" PDF, JHOVE support PDF 1.0-1.6. While PDF is backwards compatible, features introduced in

[10]http://wiki.opf-labs.org/display/Documents/Home.

[11]http://wiki.opf-labs.org/display/Documents/JHOVE+issues+and+error+messages

Table 1: Lines and Percentage of code executed by unit tests per module

Module	No. of Files	Lines of Code	Class Coverage	Module Coverage	No. of code lines in module (only *.java files)
AIFF	18	1,253	0.00%	0.00%	1,253
ASCII	1	398	0.00%	N/A%	N/A
GIF	2	60	0.00%	N/A%	60
GZIP	5	611	89.33%	90.44%	611
HTML	41	9,371	0.00%	0.00%	9,371
IFF	4	219	0.00%	38.29%	219
JPEG	9	895	0.00%	0.00%	895
JPEG2000	69	7,633	0.00%	0.00%	7,633
PDF	61	10,581	0.00%	0.00%	10,581
TIFF	61	14,457	0.87%	0.00%	14,457
WAVE	27	3,183	40.62%	6.87%	3,183
XML	9	1,498	0.00%	0.00%	1,498
Totals	309	50,705	N/A	2.97%	N/A

newer versions are currently not supported by JHOVE. The complexity of the module reflects the complexity of the PDF format itself. All Adobe PDF specifications are freely available via the company's website[12], however, as described in section 2.1, ambiguities in the specifications have lead to differing interpretations of the file format's syntactical and semantic restrictions. This complexity resulted in the PDF module consuming significant resources to complete[13] [7], and is also reflected in the continuing work on the module as described above.

But what does the module base the validation outcome on? Section 2.1 described the general two-tiered conformance approach taken by JHOVE. While the differentiation between well-formed and valid is rather straight-forward for well specified file formats such as TIFF, the situation is unfortunately more complex for the PDF file format. The module description itself states that a PDF is considered well-formed if it meets the criteria defined in Chapter 3 of the PDF Reference, breaking this down further into the following requirements [13]: "In general, a file is well-formed if it has a header:%PDF-m.n; a body consisting of well-formed objects; a cross-reference table; and a trailer defining the cross-reference table size, and an indirect reference to the document catalog dictionary, and ending with: %%EOF". Unfortunately this statement remains vague, concrete rules breaking these high-level requirements down to e.g. dictionary or object level are missing. The PDF-module documentation carries on stating that in addition to further requirements, well-formedness is a prerequisite for validity [13]. Regarding limitations of the validation by the modle, the PDF-hul documentation only states that data within content streams as well as encrypted data is not validated. While these criteria may seem straight-forward, in reality they include thousands of possibilities. Furthermore, while Adobe's specification included the "well-formedness" terminology [4], the ISO standard replaced this with "conformance"[14] [11]. The ISO standard clearly states, that it does not specify "methods for validating the conformance of PDF files", carrying on, however, by describing that "conforming PDF files shall adhere to all requirements of the ISO 32000-1 specification and a conforming file is not obligated to use any feature other than those explicitly required by ISO 32000-1." [11]. With 5.471 occurrences of the word "shall" as the ISO verbal form for a requirement and no clear differentiation between well-formed and valid, achieving rule-based conformance checking remains a lofty goal. Nevertheless, is the most suitable point of reference for what is syntactically and structural valid and what is not. Due to the complexity of both the file format and module, it should come as no surprise that validation errors like those recently fixed in JHOVE 1.16 went unnoticed for many years. However, when relying on a validation tool for digital preservation decision-making, the tool's output must be correct and complete. Ideally, a test routine exists, which checks the tool's output against a ground-truth for every new release. While such a ground-truth needs to exist on both, well-formedness and validation levels, the authors have limited the scope to criteria determining well-formedness of the object.

3 METHOD

In this section we briefly introduce software testing methods. In particular, we are highlighting the use of test corpora to measure code coverage as a form of software testing. We conclude this section with a brief description of the process used to build the light-weight test set put forth in this paper.

3.1 Software testing

JHOVE is a large established code base, currently comprising over 500 Java files and 100,000 lines of code. There are modules that validate files against twelve format specifications, each requiring its own specialist knowledge. Good automated testing is the only way to ensure that a large software project functions as expected, but how do you ascertain how well a piece of software has been tested? Before addressing JHOVE testing we define a few key software testing terms: The term *code coverage* is used to express how much of a code base has been tested. It can be measured empirically for a given test scenario by measuring how many lines of code are executed when you run it. You then divide this by the number of lines of code in the project to give a percentage figure. This task is carried out by automated coverage tools used by software testers and developers.

Testing can be carried out in different ways, one of the most effective forms is known as *unit testing*. These are small, discrete tests written by programmers when developing and fixing code. They are usually executed automatically every time that the code is

[12] Adobe Developer Connection 2017 (http://www.adobe.com/devnet/pdf/pdf_reference.html) for current version, Adobe Developer Connection Archives (http://www.adobe.com/devnet/pdf/pdf_reference_archive.html) for previous file format versions

[13] It may surprise that the original JHOVE developers stated that the complexity of the PDF-module was superseded by the HTML-module, elaborating that while other modules were designed to terminate the validation process at the first error, the HTML module typically encountered so many errors that it had to be designed to recover from errors and continue [7].

[14] Adobe: "The rules described here are sufficient to produce a well-formed PDF file" became ISO: "The rules described here are sufficient to produce a basic conforming PDF file"

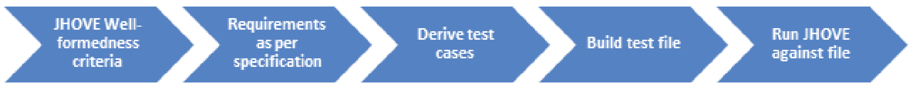
JHOVE Well-formedness criteria
Requirements as per specification
Derive test cases
Build test file
Run JHOVE against file

Figure 2: Methodology for creating and using the test set

compiled. These test the smallest components, usually a few lines of code, that make up a software package.

Integration testing is an alternative, complementary approach to unit testing. Rather than test low-level units, integration testing focuses on larger software components, including the delivered software. These tests can often take a long time to run so aren't run for every code change. Integration tests are usually run as a final test before delivering release candidate software.

Black box testing describes an approach to testing that concerns itself solely with software functionality and ignores the internal details. Nearly all software testing above the unit testing level is carried out as black box testing.

3.2 Testing JHOVE

Historically, JHOVE hasn't had a rigorous, public testing policy. There are very few unit tests, the unit testing code coverage figures for the JHOVE modules as of v1.16 are shown in Table 1[15].

For each module two coverage figures are given, each measures the percentage of code that's executed by unit tests, the first for the module control class the other for the supporting classes. The test coverage was measured using the Jacoco[16] software and its accompanying Maven[17] reporting plugins. Overall, there's less than 4% coverage of the codebase. Writing unit tests for 100,000 lines of code is at least 2 person years of effort and isn't practical given the development resources available to the project. Clearly another approach is needed to establish confidence in JHOVE's results.

3.3 Using Test Corpora and Measuring Coverage

One method suited to testing JHOVE is to create a corpus of ground truth test data designed to test the modules functionality. A good test should be atomic if possible, examining the codes handling of a specific validation issue. One large, valid PDF document might execute a high percentage of the code base without providing any insight into the manner in which JHOVE deals with validation issues at all. Currently two data sets for testing exist: The first dataset is comprised of example files that accompany the JHOVE code base. These have presumably been used by developers to test that the modules were working, without providing a formal, rigorous test corpus. The regression testing suite that the OPF is currently developing uses this data and compares the XML output of different versions of JHOVE to ensure that the results haven't changed. This is typical black box testing, treating the software as a unit that takes input files and produces XML output without worrying how the software does this. This test set comprises approximately 80 files that cover the PDF, TIFF, HTML, GIF, JPEG, JPEG 2000, XML and ASCII modules. The other test set is currently being assembled as a community effort. The aim is to gather a set of test files that between them generate every one of JHOVE's 153 PDF validation errors. First efforts, undertaken as part of the OPF DIG group and the JHOVE hack day have put forth 44 files. However, as these files come from institutions' productive archival workflows, there are some associated problems: About a third of them are subject to access regulations and may not be shared publicly. Secondly, as these are "real-world" examples, the PDF files are typically large and complex, making it hard to understand which specific part of the digital object triggered a validation rule. A ground truth test set is currently lacking.

3.4 Building a test corpus

In order to conduct straightforward black-box functional tests we introduce a set of PDFs built specifically to test the requirements of well-formedness. We limit ourselves to well-formedness, as it forms the prerequisite for a valid file. The JHOVE well-formed statement is split into the requirements for the structural sections, shown in Figure 1: header (section 4.1), body (section 4.2), cross-reference table (section 4.3) and trailer (section 4.4). For each section, the JHOVE requirement - if available - forms the starting point of the process shown in Figure 2. In a second step, the ISO 32000-1:2008 standard is checked to transform the high-level criterion into individual requirements / test cases. These requirements are then implemented as a test file, which is validated using JHOVE and rendered with Adobe Acrobat Professional. To produce atomic tests against syntactical file format requirements, a minimal well-formed and valid PDF ("Hello_World.pdf") was created and used as the basis for all other test files. This file consists of a single page which includes one font definition as a resource and a single text stream as a content. The graph including respective object Ids is shown in Figure 3. Information on test cases and accompanying test results are captured in a google spreadsheet.

4 TEST CORPUS

This synthetic test corpus consists of 90 files, which are available on the JHOVE github repository [25].The test set contains the minimal PDF file (minmal_test.pdf) given as an example in section 2.1, the starter file "Hello_World.pdf" as well as 88 test cases which are derivations of "Hello_World.pdf". Test cases were created using the

[15]See CodeCov results page for full coverage results: https://codecov.io/gh/openpreserve/jhove/tree/0093b7dd74d3d7582eff0d8f2e22eb7a89f5befd/jhove-modules/src/main/java/edu/harvard/hul/ois/jhove/module
[16]http://www.eclemma.org/jacoco/
[17]http://www.eclemma.org/jacoco/trunk/doc/maven.html

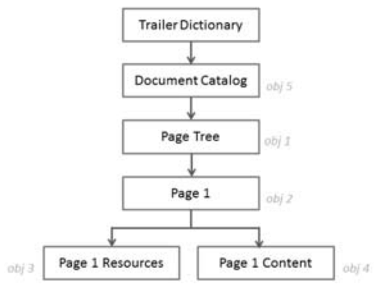

Figure 3: Graph and according PDF object Ids for Hello_World.pdf file

process described in section 3.2. To allow files within the test corpus data set to be referenced, we introduce a naming scheme for the test files and cases. The scheme is based on a scalable ontology, which follows the basic PDF structure as shown in Figure 1. The main four sections are numbered with the body section (T02) branching off in separate subsections for possible object types. For this paper, the object types document catalog (T02-01), page tree node (T02-02), page node (T02-03), page resource (T02-04) and stream object (T02-05) are analyzed further. While different types of objects are possible for page resources and stream objects, we only focus on font (T02-04-01) and text respectively. Each test case is numbered according to the section in which the deviation is introduced in (Txx-xx-xx), followed by a 3 digit number (_xxx). In addition to the test case ID, the file names contain a brief description of the feature tested, e.g. "T001_header_invalid-major-version.pdf". All of the test files are listed in a google spreadsheet[18]. The created test corpus has been made available on GitHub[19].

4.1 PDF Header

The PDF-hul documentation prescribes that the PDF Header consists of the first line of the file which must contain the five characters "%PDF-", followed by a version number. According to ISO32000-1:2008, the version number is of the form 1.N, where N is a digit between 0 and 7 [11] - Adobe's documentation states simply that the syntax is %PDF-M.m., where M is the major and m the minor version [3]. In addition to the general syntax, each Adobe specification explicitly names the header relevant for the respective format version, e.g. %PDF-1.6 for PDF 1.6 [4]. Beginning with PDF 1.4 the version may also be included in the document's catalog dictionary. If present, it shall be used instead of the version in the header. While this information is relevant for adequate identification of the file format version, it does not replace the structural requirement of the header.
Test cases are based on the syntax requirement as well as by using plausible version numbers. Deviations from the syntax are added to the first 5 chars %PDF- as well as to the version notation M.m. An additional test case is formed by removing the header. In total 7 test cases were created for header deviations, 1 missing its header (T01_007), 2 containing invalid major / minor versions (T01_001 - 002) and 4 with syntactical header errors (T01_003 - 006)./newline The file with a missing header and the 4 objects with syntactical errors were successfully detected by JHOVE as not being well-formed. For all test objects considered *Not well-formed*, PDF-hul returned the error "No PDF header". It is debatable whether this error description is correct as the header does exist but contains invalid information. JHOVE did not handle major and minor versions test cases as well. Instead of either limiting the possible version entries to the profiles the module supports or to all possible combinations of M.m, JHOVE only checks against M=1. This leads to the non-existing version 1.9 being accepted as *well-formed and valid* (T01_002).

4.2 PDF Body Well-formed objects

The JHOVE PDF-hul module requires that "the file has a body, consisting of well-formed objects". PDF supports five basic and three compound objects: Integers / real numbers; strings which must be enclosed in parentheses; names, which are introduced by a forward slash; boolean values; the null object, denoted by keyword null; arrays consisting of an ordered listing of objects, e.g., [1 0 0 0]; dictionaries and streams [33]. Objects are linked to each other via indirect references. Additionally, PDFs are divided into nodes as objects (obj). The content of these nodes, such as a page tree node, are regulated in the standard, perscribing required and optional objects within. This paper focuses on the five PDF node structures required to construct a minimal PDF as shown in Figure 3: the document catalog (4.2.1), the page tree node (4.2.2), the page node (4.2.3), the page resource node containing the resource font (4.2.4) and the stream node containing text (4.2.5).

4.2.1 Document catalog. The JHOVE well-formedness description only mentions the document catalog in conjunction with the trailer (see 4.4), which must contain an indirect reference to the document catalog. However, the document catalog is the root of the PDF's object hierarchy graph (see Figure 3), containing references to all other objects which define the document's content, outline, threads and attributes. The document catalog can also contain information about how the document shall be displayed, e.g., defining a default page other than page 1 to first show when the file is rendered. While a number of key pairs are possible within the catalog dictionary, only two are required: *Type* with value *Catalog*, which defines the object type of the dictionary, and *Pages*, which contains an indirect reference to the page tree node [11]./newline The test cases present a missing or inaccessible *document catalog* (T02-01_001 - 002), a missing or incorrectly defined *Pages* indirect object (T02-01_003 - 004) and a missing *Type* key value pair (T02-01_005 - 007)./newline Analyzing the results of running JHOVE across the test set revealed some troubling behavior. While JHOVE correctly recognizes when the document catalog is removed completely, it does not appear to cross-check the indirect reference in the trailer and the actual object number of the document catalog. While the file (T02-01_002) is not well-formed according to the standard and cannot be rendered by Adobe Acrobat, JHOVE reports the file as being *well-formed and valid*. The suspicion that referenced object numbers are not cross-checked against the target objects is

[18]Lindlar, Tunnat, Wilson: Test corpus description and outcome - available at https://docs.google.com/spreadsheets/d/1SWa2MtiSUQDVmlBvGb2a-b_zn7SmARv_CERXlFFDED0/edit?usp=sharing
[19]https://github.com/openpreserve/jhove/tree/ipres/pdf-test-all/test-root/corpora/ipres-paper-pdfs/modules/PDF-hul

further confirmed by the test case containing an incorrect indirect reference in the *Pages* value (T02-01_004). Here, the file is also flagged as *well-formed and valid* by JHOVE but cannot be rendered by Adobe Acrobat, as it fails to locate the page tree node needed to display the pages. Similarly, the requirement that the value for *Type* must be *Catalog* is not checked. Invalid entries are considered *Well-formed and valid* by JHOVE. However, as opposed to the other false negatives, which could not be rendered by the reader software, the test file with an invalid *Type* key value (T02-01_006) renders correctly and without a warning.

While the validation against the values result in incorrect outcomes, the routines implemented in JHOVE do correctly check the existence of the required key values *Pages* and *Type* - the absence of which results in objects being reported as *Not well-formed* with "Invalid object definition" error messages (T02-01_003, 005, 007).

4.2.2 Page tree node. While the JHOVE well-formedness requirements do not mention the page tree object, the object is required in order to interpret and access the file's content correctly. Every PDF object has at least one page tree node, which forms the basis of the hierarchical structure of page leaf nodes and/or further page tree nodes. Required information within the page tree node dictionary is the dictionary *Name* with required value *Pages*, the number of children the node has (further page tree nodes or pages) given in the *Count* key value pair, and lastly an array of indirect objects of the children in the *Kids* key value pair. If the page tree node is not the root page tree node, i.e., not the first page tree in the document, a *Parent* key value entry is also required. As our lightweight test set only contains a single page tree node, this value is not required and is not checked within the test set.

The test cases present a missing page tree node (T02-02_001), missing or malformed *Type* (T02-02_006, T02-02_009) and *Count* (T02-02_007 - 008) key value pairs and a missing or malformed *Kids* array (T02-02_003 - 005).

The result of running JHOVE across this test set is similar to that of the previous section - a missing page tree node is detected by JHOVE, resulting in a *Not well-formed* object and an error message stating "Invalid object definition". The same result - *Not well-formed* and "Invalid object definition" - is obtained when validating the test cases with missing required keys *Type* (T02-02_006) and *Count* (T02-02_008). In both cases the file renders correctly in Adobe reader, however, Adobe attempts to fix the error and prompts the user to save the changes upon closing the file. As in the case of the document catalog key value pairs, the values for *Type* and *Count* do not appear to be checked by JHOVE - invalid values such as an incorrect integer value for *Count* result in *well-formed and valid* files according to JHOVE despite the fact that they clearly violate the requirements defined in the standard.

In the case of the array *Kids*, values appear to be at least partially checked, invalid indirect objects such as a reference to self (T02-02_002) and no kids (T02-02_005) or a combination of existing and non-existent children (T02-02_003) are caught correctly, resulting in a not-well formed outcome and error messages such as "Excessive depth or infinite recursion in page tree structure" for self-reference or "Invalid object definition" for missing kids. The test case containing only one, non-existent object as the only array entry for the *Kids* key (T02-02_004) was interesting, as it resulted in a *well-formed, but not valid* outcome. It is unclear, why this is considered well-formed as opposed to, e.g., the combination of valid and invalid array entries found in test case T02-02_003 which are flagged as *not well-formed* by JHOVE.

4.2.3 Page node. No specific requirements concerning page objects are given in the JHOVE PDF-module description, however, per standard a PDF file must have at last one page object. The page object dictionary can contain 30 different key pair values, but only 4 of them are required: *Type*, *Parent*, *MediaBox* and *Resources*. *MediaBox* and *Resources* can be inherited from ancestor nodes in the page tree. Other keys are only required under specific conditions, e.g., *StructParents* is required, if the PDF contains structural content items. It's interesting to note that *Contents* is not a required key value pair. If no *Content* object is referenced, the page is simply blank. As our sample file contains one *Contents* object, we are testing the *Contents* reference despite the fact that it is only an optional key.

The test cases present a missing page object (T02-03_006), and missing or malformed *Type* (T02-03_001, T02-03_002), *Parent* (T02-03_003 - 005), *Resources* (T02-03_008, T02-03_012), *MediaBox* (T02-03_008 - 009) and *Contents* (T02-03_010 - 011) key value pairs.

As in the analysis for the other body test cases, a missing node (T02-03_006) as well as missing key value pairs (T02-03_001, T02-03_003, T02-03_008, T02-03_012) result in *Not well-formed* files with "Invalid Object definition" errors. This includes the test cases where the *Contents* key value pair is missing (T02-03_010) - an interesting result, considering that the *Contents* key value pair is optional according to the standard. An absence of *Contents* should therefore result in an invalid, but not directly in a not-well formed status. Moreover, an invalid value entry for *Contents* results in a *well-formed, but not valid* output (T02-03-011), which is surprising as in previous dictionary cases plausibility and correctness of values was rarely checked and never lead to *well-formed, but not valid* outputs. The respective test file cannot be rendered by Adobe Acrobat and leads to the rendering application crashing - clearly not the result we expected for a *well-formed but not valid* object. Indirect values for *Parent* (T02-03_004) and *Resources* (T02-03_007) go unnoticed, resulting in *Well-formed and valid* outputs. Checking against inconsistencies for the *Type* values (T02-03_002) lead to an interesting discovery. After having changed the *Type* value from the required *Pages* to *Catalog*, JHOVE returned the file as *Not well-formed.* This was surprising, as the previous test cases had returned invalid entries for the Type key value pair as *well-formed and valid.* Further analysis revealed that when the Type was then changed to Font and re-validated the outcome was *Well-formed and valid*, even though the value was still wrong. This leads to the assumption that some but not all values for *Type* are checked and the value *Catalog* always leads to further analysis by the software. Further test cases which were handled correctly by JHOVE's validation routine and were identified as *Not well-formed* are a wrong object type for the *Parent* value, which expects a single indirect reference (T02-03_005) and the wrong number of parameters for the *MediaBox.*

4.2.4 Page Resource - Font. Well-formedness criteria for page resources in general and fonts in particular are not addressed by the description of JHOVE well-formedness criteria. As per standard, resources for a page, such as images or font, may be described in resource dictionaries which can be included in dedicated page resource objects. Alternatively, resources may also be directly described within content stream objects. This paper only briefly examines page resources for fonts. The use of fonts in PDF is a particularly complex subject [5], hence this is only a high level analysis focusing on font dictionaries and associated data structures. This includes a look at the information a conforming reader requires to interpret the text and position the glyphs correctly. Required key value pairs in the font dictionary are *Type*, *Basefont* and *Subtype*. The test cases present a missing or invalid resource object (T02-04-01_003, T02-04-01_004), a missing or malformed *Subtype* (T02-04-01_005, T02-04-01_006) and missing *BaseFont* (T02-04-01_002, T02-04-01_006) key value pairs. As is the case with other tests, a missing dictionary *Type* key value pair (T02-04-01_003) resulted in a *not well-formed* JHOVE result, while an invalid value - in this case *Page* instead of the expected *Font* (T02-04-01_004) - produces a *well-formed and valid* result. The *Subtype* key identifies the font type, ISO32000 lists the following valid values: *Type0, Type1, MMType1, Type3, TrueType, CIDFontType0* and *CIDFontType2*. A missing *SubType* key value pair (T02-04-01-005) from the font dictionary results in an unrenderable file, which JHOVE correctly catches as *Not well-formed* with an "invalid object definition error". A more distinct error message indicating the magnitude of the error would be helpful here. A wrong value for *SubType* (T02-04-01_006) unfortunately goes unnoticed - resulting in a *well-formed and valid* JHOVE result. The *BaseFont* key contains the actual font - for *Type1* fonts this is typically the name it is known by in the respective font program. A missing *BaseFont* (T02-04-01_001) results in Adobe being unable to render any text which uses the font. JHOVE catches this error as a well-formedness violation. Again, the error message is "invalid object definition" as well as an additional "no document catalog dictionary" error. This appears to be a resulting downstream error in parsing dictionaries and is misleading here. A *BaseFont* with the wrong value (T02-04-01_002) also results in a "no document catalog dictionary" error, which is again misleading. Adobe Reader renders the text using an alternative, default font.

4.2.5 Stream Objects - Text. JHOVE states no specific requirements for stream objects. Streams are used to store binary data or text to be displayed on a page. For this paper, we only examine a simple, uncompressed text stream. As per ISO standard, the requirement for printing text on page is a *Stream* dictionary including the *Length* key value with the number of bytes between the *stream* and the *endstream* keywords. The stream dictionary must be followed by a descriptor stating the position of the text on the page, a beginning text object marker (*BT*), operators to choose the text font (*Tf*) and size, the text itself, the font show operator (*Tj*), the end text marker (*ET*) and the *endstream* keyword. The standard mandates that no extra bytes other than white space are allowed between the *endstream* and the *endobj* markers. The test cases check the presence of the required text operators (T02-05-01_001 - 008, T02-05-01_012), correct syntax of the text object (T02-05-01_009 - 011), missing keywords *stream* (T02-05-01_13) and *endstream* (T02-05-01_14), missing or invalid *Length* key value pair (T02-05-01_015, T02-05-01_016) and extra bytes between *endstream* and *endobj* keywords (T02-05-01_017). Running JHOVE across the test files showed that all text operators are checked by the PDF-module. The absence of the operators is detected, resulting in a *Not well-formed* output (T02-05-01_001 - 008, T02-05-01_012). As these errors typically result in the rendering application being unable to open the file, it is particularly important that JHOVE detects them. However, the accompanying error message for the test cases: "No document catalog dictionary", appears to be a down-stream error and does not indicate that the problem is in the stream object or more specifically in a missing text operator, which would be important information for the user. The PDF-module correctly detects missing *stream* (T02-05-01_13) and *endstream* (T02-05-01_14) keywords and missing or invalid *Length* key value pairs (T02-05-01_015, T02-05-01_016). Validation issues were detected when processing the actual text streams. String objects can be either encoded as literal strings enclosed in parentheses, or as hexadecimal streams enclosed in angle brackets [11]. Our lightweight test set includes a literal string. However, missing opening or closing parentheses (T02-05-01_009, T02-05-01_010) as well as a substitution with brackets (T02-05-01_011) goes unnoticed by JHOVE, returning a *Well-formed and valid* result. This is especially grave as the reader fails to render the files, showing the message "An error exists on this page. Acrobat may not display the page correctly. Please contact the person who created the PDF document to correct the problem".

4.3 Cross reference table

The cross reference table enables random access to the various objects contained in a PDF and is an essential element of any PDF file. JHOVE acknowledges this, the mandatory presence of the cross reference table is mentioned in the well-formedness conformity statement. However, as with other objects, JHOVE gives no further requirements for the table. While conforming PDF implementations may divide information between multiple cross-references streams, cross-reference sections and cross-reference tables, this paper only examines a lightweight PDF with a single cross-reference table.
Per standard, the cross-reference section must start with the *xref* keyword. If the file only contains one table and has never been updated, as is the case with our test file, the second line should start with 0 and include a second number stating the number of entries in the table - in our case 6. Finally, the table contains one entry for each object. Each entry is exactly 20 bytes in length and consists of the off-set (10-digit), the generation number (5-digit) and a keyword indicating whether the object is in use (*n*) or free (*f*). [11]
The test cases present a missing cross-reference table (T03_001), missing *xref* keyword (T03_002), missing or invalid number of entries (T03_003, T03_004), missing or invalid offsets (T03_006 - 008), invalid entry keywords (T03_009) and invalid generation numbers (T03_010).
The test files for missing cross reference table and *xref* keyword, (T03_001, 002) as well as for an invalid number of entries (T03_003 - 005) were detected correctly as *Not well-formed.* An interesting outcome in this context was a software bug when performing the test against missing number of entries (T03_003), producing the following error: "edu.harvard.hul.ois.jhove.module.pdf.Keyword cannot be cast to edu.harvard.hul.ois.jhove.module.pdf.Numeric". This is a Java exception thrown when the application has tried to

perform an undefined data conversion.
Only one test case for the cross-reference table led to an incorrect validation result, (T03_010) - here, an invalid generation number for an entry was proclaimed as *Well-formed and valid* by JHOVE.

4.4 PDF Trailer

The PDF-module description states that to be well-formed, a file must have "a trailer defining the cross-reference table size with an indirect reference to the document catalog dictionary, and ending with: %%EOF" [13]. The PDF standard is more specific in its requirements, stating that the trailer must start with the trailer dictionary, consisting of the *trailer* keyword and key-value pairs enclosed in double angle brackets. Two key value pairs are required for all PDFs: *Size* of type integer, which holds the total number of entries in the cross-reference table, and *Root* of type dictionary, which contains the indirect reference to the catalog dictionary or root object of the PDF. Other key value pairs are reserved for particular PDF properties, such as *Encrypt* and *ID* for encrypted documents, as well as *Prev* for objects with more than one cross-ref section. The trailer dictionary is followed by the *startxref* keyword and the byte offset counting from the beginning of the file to the last cross-reference section. The trailer and the object closes with the end-of-file marker *%%EOF* [11].
For the test cases, The JHOVE well-formed criteria stated above are broken down into the existence of a properly formed trailer (T04_008 - 010), the existence and validity of the mandatory key value pairs *Size* (T04_015, T04_016) and *Root* (T04_011 - 014). The last line of a PDF file contains only the end-of-file-marker *%%EOF*. As PDF files are typically read starting with the trailer, the EOF-marker is an essential keyword, indeed most applications will not parse or render the file if it is missing [3]. The test set contains a number of invalid variations of the *%%EOF* tag (T04_001 - 007).
While not explicitly mentioned in JHOVE's conformance requirements, the offset of the cross-reference section prefaced by the *startxref* keyword is an essential part of the trailer. Because of this, offset and keyword are also included in the test set (T04_018 - 019).
With one exception, every test case pertaining to the trailer dictionary or the cross-reference table byte offset produced the expected status message *Not well-formed*. The exception was an unexpected program termination in the test case which omitted the closing brackets. It seems as if the module gets stuck in an endless loop.
Another unclear case were two error messages which appear to be incomplete ("4" and "Null"). One arose in the case of an incorrect value of type integer given in *Size*, i.e., not the correct size of the cross-reference table entries. Here, one of the errors simply returns the value "4". Further experiments have shown if the number of entries does not equal the value of *Size*, the error message returns the value stated in the *Size* key value pair. This should be replaced with a more detailed message. The "Null" error appears to have similar origins, as it was produced by the test case with a missing *Size* entry. Two of the test cases produced validation errors rather than well-formedness violations: "Size entry missing in trailer dictionary" and "Root entry missing in trailer dictionary". While both test cases result in a *Not well-formed* status due to a subsequent error ("No document catalog dictionary"), the missing entries should be reason enough to fail the syntactical check, as the standard clearly states that *Size* and *Root* keys are mandatory elements of the trailer dictionary. Further dictionary errors thrown, such as "Malformed indirect object reference" or "Improperly nested dictionary delimiters" are generic to all dictionaries and also appear in other objects.
The main problem found when validating the *%%EOF* tag test cases was that JHOVE is tolerant towards data being present after the *%%EOF* tag. Following incremental updates a PDF file might contain several *%%EOF* tags, still the last line must be *%%EOF*. The test case containing junk data after the tag (T04_005) passed JHOVE validation as *well-formed and valid*. Furthermore, the ISO standard states that the *%%EOF* tag should be present in the last line of the file. JHOVE validation simply follows the requirement that %%EOF is the last string in the file, regardless of it being on a line of its own (T04_001), resulting in a *well-formed and valid* file.

5 DISCUSSION

As part of the work presented in this paper we have developed a light-weight test set for JHOVE's PDF validation routine against well-formedness requirements derived from the ISO 32000-1:2008 standard for PDF. As presented in section 1, our aim has been to:

(1) establish a ground truth for well-formedness criteria
(2) test the JHOVE software against that ground truth; and
(3) improve automated regression testing

Within this section we will briefly discuss if and how the test set meets these goals.

5.1 Establishing a ground truth

The JHOVE PDF-module's description of the requirements which need to be fulfilled to be considered well-formed are fairly vague. Particularly the definition of what comprises a well-formed body is high level. Neither a ground truth test corpus nor a clear overview of the validation criteria enforced by the JHOVE PDF-module are available.
In this paper, we have checked the criteria presented in the PDF-module documentation against the concrete requirements stated in the PDF ISO 32000 standard. Using a light-weight test object, consisting of a minimal set of structural objects, we have produced ground truth data for a small sub-set of criteria for the validation against structural and syntactical requirements. As the ISO standard does not include a differentiation between well-formed and valid, we have defined well-formed as the required syntactical and structural aspects of the PDF graph in general and the object's used within in particular. This approach shows that the line between well-formed and valid for PDF is unfortunately by no means as straightforward as the XML example included in section 2.1. This will be especially challenging when the work described here is taken forward, looking at the optional structural elements of PDF, such as object requirements for linearized PDFs, which have fixed requirements in themselves. Is a linearized PDF only well-formed when all requirements for linearization are met? Or is any requirement violation occurring in an optional structural part only a violation of validity?

5.2 Testing JHOVE against ground truth

The test cases included in this paper have shown clearly how error-tolerant rendering software can be. Files syntactically wrong at

very elementary levels may happily be rendered by tools such as Adobe Acrobat without a warning. However, we've also produced examples where the reader failed to render objects which JHOVE deemed to be *well-formed and valid*, further underlining the invaluable asset of a ground truth test set. By running the test cases against the JHOVE PDF-module, we have discovered a number of inconsistencies between expected outcome for a test-case and the de-facto validation result returned by JHOVE 1.16 / PDF-hul 1.8. These inconsistencies are being raised as issues on the GitHub project site where they will be picked up by the JHOVE maintainers. As a first step, we have opened issues on the JHOVE GitHub repository for 9 of the discovered inconsistencies:
issue #207: PDF version checking incorrect
issue #208: Inconsistent catalog indirect reference and object number
issue #209: Inconsistent Pages indirect reference and object number
issue #210: Value for Type in Document Catalog not validated
issue #211: Indirect reference to not existing object in page tree node Kids array results in *well-formed and valid*
issue #212: Value for Type in Page Tree Node not validated
issue #213: Consistency between /Parent and Kids for Page Tree Node and Page Object not checked
issue #214: Paranthesis around literal strings are not checked
issue #215: Error message - JHOVE expects integer, gets string in cross-reference stream with missing value

5.3 Improving test coverage and regression testing

A corpus based testing methodology has been described in section 3.2. In this section we examine the utility of the test set produced with this paper as a test corpus for JHOVE PDF validation. This was tested by measuring and analyzing the coverage for the individual files and the test set as a whole.
The "hello_world.pdf" seed file covered 36% of the code in the PDF module's control class and 26% of the module code. Examining the coverage figures reveals that when the problems presented in the test files were undetected by JHOVE the coverage figures for the test are identical to the seed file. That makes sense as the application found no problems in the file so continued processing the file resulting in more of the code being executed. Because of this we've omitted these results from our summary as they obscure the coverage figures for the discrete tests that fail as expected. Table 2 shows the coverage figures produced by running JHOVE over the test classes described in this document.

Regarding these test figures, it's worth noting the manner that the combined coverage figures are never higher than around 36% for the controlling class and 26% for the module. Furthermore these figures are identical to the coverage generated by the seed file. This means that many of the tests are exercising the same areas of the code base reflecting similarities in the test cases themselves. It's important not to get too carried away with test coverage in this respect. The real use of these test file is not the amount of code they execute but in which parts of the code they execute. Synthetic test corpora can be crafted to exercise specific sections of the code base. Initially this can be done by studying JHOVE's validation criteria and the PDF standards as is the case for the work presented here.

Table 2: Class and module coverage of test case files presented in this paper

Test File	Class Coverage	Module Coverage
T01 Header tests	9%	17%
T02-01 Document Catalog tests	21%	18%
T02-02 Page Tree tests	21%	20%
T02-03 Page Object tests	26%	22%
T02-04 Resources tests	19%	18%
T02-05 Stream tests	19%	18%
T03 Cross Reference tests	28%	24%
T04 Trailer tests	19%	20%
All Test Files	36%	26%

Table 3: Module coverage of corpus presented in this paper compared to other corpora

Corpus	All Modules coverage	PDF Module Coverage
Synthetic corpus for this paper	13%	26%
JHOVE example files	51%	50%
JHOVE error corpus	18%	49%
All Test corpora	54%	58%

As test coverage increases it's possible to see which areas of the code remain untested and use this as another guide when creating test files.
Even after JHOVE validation has been tested and verified to function as expected the test corpus has a vital role in ensuring that this remains the case. The data set can be used to regression test new releases of JHOVE to make sure that fixed bugs are not inadvertently reintroduced as new development takes place. Finally we'll examine coverage figures for the two other test data sets introduced in section 3.3 - the intital JHOVE example files and the OPF JHOVE error corpus - and compare the figures to this corpus for context.

Due to the complexity of the files within the other corpora, they produce greater coverage of the PDF module's code. However, the high coverage comes as a price: due to their complexity, they are also of less use in identifying and fixing problems. For complex, "real-world" files it is much harder to predict which areas of the code they will test, in turn making it harder to diagnose problems as there's more code to analyze. Furthermore, many bugs can be caused by interactions between features that weren't properly considered when developing the software. In summary synthetic test files are ideal for testing how the software deals with the individual elements of the specification. Real world files are excellent candidates for testing software's function when presented with more complex files as well as measuring performance but only after confidence in the software has been established through formal testing using synthetic test data.

6 SUMMARY

In the work presented in this paper, the authors have created a light-weight test set for the validation of PDFs at the structural well-formedness level. The test set consists of 90 files and 88 test cases and is publicly available via the JHOVE GitHub repository[20]. Additionally, the outcome of the validation and rendering tests described in this paper as well as the detailed figures for the code coverage of the test set in regards to the JHOVE PDF-module are described in a spreadsheet available online[21].

The authors have shown how the test corpus can be used to serve three purposes: to establish a ground truth for what is not well-formed, to test the JHOVE software against that ground truth and lastly to improve automated regression testing. Inconsistencies discovered in running the ground truth data against JHOVE are being fed back into the development process via GitHub issues. Furthermore, the test set and process will be shared with the JHOVE community, hoping to stimulate discussion around the methodology used and triggering further efforts in extending the test data to cover more features of PDF files and other format modules.

If we want the software we use to be fit for the lofty, long term goals that the digital preservation community aspires to it needs to be tested thoroughly. This testing needs to be public and demonstratively complete, and who better to make sure that this is the case than the community who use the software. Only when this testing is in place will the JHOVE status of *Well-formed and valid* carry the lifetime guarantee we want it to.

REFERENCES

[1] Stephen Abrams. 2007. *Instalment on "File Formats". DCC Digital Curation Manual.* Technical Report. DCC Digital Curation Centre. https://www.era.lib.ed.ac.uk/bitstream/handle/1842/3351/Abrams%20file-formats.pdf

[2] Stephen Abrams, Sheila Morrissey, and Tom Cramer. 2009. "What? So What": The Next-Generation JHOVE2 Architecture for Format-Aware Characterization. *International Journal of Digital Curation* 4, 3 (2009), 123–136. http://www.ijdc.net/index.php/ijdc/article/view/139

[3] Adobe Systems Incorporated. 2001. *PDF reference : Adobe portable document format version 1.4 / Adobe Systems Incorporated. - 3rd ed.* ADDISON-WESLEY. http://www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/pdf_reference_archives/PDFReference.pdf

[4] Adobe Systems Incorporated. 2006. *PDF Reference - sixth edition: Adobe Portable Document Format Version 1.7.* Technical Report. Adobe Systems Incorporated. http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf

[5] Ahmet Cakir. 2016. Usability and accessibility of portable document format. *Behaviour & Information Technology* 35, 4 (2016), 324–334. DOI: https://doi.org/10.1080/0144929X.2016.1159049 arXiv:http://dx.doi.org/10.1080/0144929X.2016.1159049

[6] COPTR. 2017. JHOVE2. COPTR Preservation Registry entry. (2017). http://coptr.digipres.org/JHOVE2.

[7] Martin Donnelly. 2006. *DCC Digital Curation Centre Case Studies and Interviews: JSTOR/Harvard Object Validation Environment (JHOVE).* Technical Report. Digital Curation Centre. https://www.era.lib.ed.ac.uk/bitstream/handle/1842/3335/Donnelly%20jhove.pdf

[8] G. Endignoux, O. Levillain, and J. Y. Migeon. 2016. Caradoc: A Pragmatic Approach to PDF Parsing and Validation. In *2016 IEEE Security and Privacy Workshops (SPW).* 126–139. DOI: https://doi.org/10.1109/SPW.2016.39

[9] Peter Fornaro and Lukas Rosenthaler. 2016. Long-term Preservation and Archival File Formats: Concepts and solutions. *Archiving Conference* 2016, 1 (2016), 87–90. DOI: https://doi.org/doi:10.2352/issn.2168-3204.2016.1.0.87

[10] Yvonne Friese. 2014. Ensuring long-term access: PDf validation with JHOVE? PDF Association Blog Post. (2014). https://www.pdfa.org/ensuring-long-term-access-pdf-validation-with-jhove/

[11] ISO/TC 171/SC 2. 2008. *ISO, ISO 32000-1:2008, Document management – Portable document format – Part 1: PDF 1.7.* International Organizsation for Standardization. https://www.iso.org/standard/51502.html

[12] Andrew N. Jackson. 2008. Does JHOVE Validate PDF/A Files? Blog Post. (2008). http://anjackson.net/keeping-codes/experiments/does-jhove-validate-pdfa-files.html

[13] JSTOR. 2006. JHOVE PDF-hul. Online. (2006). http://jhove.sourceforge.net/pdf-hul.html

[14] JSTOR. 2006. JHOVE TIFF-hul. Online. (2006). http://jhove.sourceforge.net/tiff-hul.html

[15] Amy Kirchhoff and Sheila M. Morrissey. 2016. *Digital Preservation Metadata Practice for E-Journals and E-Books.* Springer International Publishing, Cham, 83–97. DOI: https://doi.org/10.1007/978-3-319-43763-7_7

[16] Michelle Lindlar and Yvonne Tunnat. 2017. How valid is your validation? A closer look behind the curtain of JHOVE. In *12th International Digital Curation Conference: Upstream, Downstream: embedding digital curation workflows for data science, scholarship and society.*

[17] Peter May. 2017. Testing JHOVE PDF Module: the good, the bad, and the not well-formed. OPF Blog Post. (2017). http://openpreservation.org/blog/2017/03/10/testing-jhove-pdf-module-the-good-the-bad-and-the-not-well-formed

[18] Gary McGath. 2014. Mavenized JHOVE. Mad File Format Science blog post. (2014). https://madfileformatscience.garymcgath.com/2014/03/

[19] Gary McGath. 2015. File identification tools, part 9: JHOVE2. Mad File Format Science blog post. (2015). https://madfileformatscience.garymcgath.com/2015/07/14/fident-9/#more-1578

[20] Gary McGath. 2015. A new home for JHOVE. Mad File Format Science blog post. (2015). https://madfileformatscience.garymcgath.com/2015/02/03/newjhove/

[21] Gary McGath and Carl Wilson. 2016. JHOVE File validation and characterization. Online Resource. (2016). https://sourceforge.net/projects/jhove/

[22] Becky McGuinness. 2016. JHOVE Online Hack Day Report. OPF Blogpost. (2016). http://openpreservation.org/blog/2016/10/19/jhove-online-hack-day-report/

[23] Open Preservation Foundation. 2015. *Digital Preservation Community Survey 2015.* Technical Report. Open Preservation Foundation. http://openpreservation.org/public/OPFDigitalPreservationCommunitySurvey2015.pdf

[24] Open Preservation Foundation. 2015. *JHOVE Evaluation & Stabilisation Plan.* Technical Report. Open Preservation Foundation. http://openpreservation.org/public/OPF_JhoveEvaluationStabilisationPlan.pdf

[25] Open Preservation Foundation. 2017. JHOVE. GitHub Repository. (2017). https://github.com/openpreserve/jhove

[26] Open Preservation Foundation. 2017. JHOVE: Open source file format identification, validation & characterisation. Website. (2017). http://jhove.openpreservation.org/.

[27] Lavdrim Shala and Ahmet Shala. 2016. File Formats - Characterization and Validation. *IFAC-PapersOnLine* 49, 29 (2016), 253 – 258. DOI: https://doi.org/10.1016/j.ifacol.2016.11.062

[28] Yvonne Tunnat. 2016. Error detection of JPEG files with JHOVE and Bad Peggy - so who's the real Sherlock Holmes here? OPF Blog Post. (2016). http://openpreservation.org/blog/2016/11/29/jpegvalidation/

[29] Yvonne Tunnat. 2017. TIFF format validation: easy-peasy? OPF Blog Post. (2017). http://openpreservation.org/blog/2017/01/17/tiff-format-validation-easy-peasy/

[30] Johan van der Knijff. 2017. Breaking WAVEs (and some FLACs). OPF Blog post. (2017). http://openpreservation.org/blog/2017/01/04/breaking-waves-and-some-flacs/

[31] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* Technical Report. W3C. https://www.w3.org/TR/REC-xml/

[32] Simon Whibley and et al. 2016. *WAV Format Preservation Assessment.* Technical Report. British Library. http://wiki.dpconline.org/images/4/46/WAV_Assessment_v1.0.pdf

[33] John Whitington. 2011. *PDF Explained.* O'Reilly Media.

[20] https://github.com/openpreserve/jhove/tree/ipres/pdf-test-all/test-root/corpora/ipres-paper-pdfs/modules/PDF-hul

[21] https://docs.google.com/spreadsheets/d/1SWa2MtiSUQDVmlBvGb2a-b_zn7SmARv_CERXlFFDED0