SCAPE
Scalable Preservation Environments

Evaluation of characterisation tools

Part 1: Identification

Authors

Johan van der Knijff, KB/National Library of the Netherlands

Carl Wilson, British Library

This work was partially supported by the SCAPE Project. The SCAPE project is co-funded by the European Union under FP7 ICT-2009.4.1 (Grant Agreement number 270137).

Table of Contents

1 Introduction

This report is part of SCAPE Work Package 9 – Characterisation Components. The overall objective of this work package is the development of a technical infrastructure that enables large scale characterisation of digital objects in a distributed architecture. A first step towards this end is an analysis of existing tools. What tools are available, what are they capable of, and to what extent can they be deployed in the envisaged architecture?

1.1 Scope of this document

In practice the term "characterisation" is often used to indicate quite different things. To avoid any confusion, this document follows the terminology used in the JHOVE2 project. Here, characterisation is loosely defined as the process of deriving information about a digital object that describes its character or significant nature[1]. This process is subdivided into four aspects:

- *identification* - the process of determining the presumptive format of a digital object;
- *feature extraction* - the process of reporting the intrinsic properties of a digital object that are significant to preservation planning and action;
- *validation* - the process of determining the level of conformance of a digital object to the normative syntactic and semantic rules defined by the authoritative specification of the object's format;
- *assessment* - the process of determining the level of acceptability of a digital object for a specific purpose on the basis of locally-defined policy rules.

Although the characterisation work in SCAPE will eventually cover all the above aspects of characterisation, the scope of this document is limited to identification only. The selection of tools is based on an inventory that is covered in detail in a separate document (van der Knijff *et al.*, 2011b).

1.2 Outline

The evaluation of the identification tools is based on an evaluation framework. Chapter 2 gives a brief explanation of the evaluation criteria. A detailed description of the evaluation framework is given in a separate document (van der Knijff *et al.*, 2011a).

Chapter 3 gives a description of the data sets that were used for the evaluation, and of the technical environment in which the tests were run.

The actual tool evaluations are covered in Chapters 4 to 8. The currently evaluated tools are:

- DROID 6.0
- Fido 0.9
- Unix File tool
- FITS 0.5
- JHOVE2

Note that some of these tool offer functionality that extends well beyond identification. This applies in particular to FITS and JHOVE 2, which can also be used for feature extraction, validation and assessment. However, the scope of the current report is limited to identification only, and these additional functionalities were not included in the evaluation.

[1] JHOVE2 actually uses 2 separate definitions: "(1) Information about a digital object that describes its character

An important limitation of the current work is that the quality of the identification results has not been taken into account, even though this is part of the evaluation framework. The main reason for this is the lack of a reliable test corpus, in which the ‘true’ format of each file is unambiguously known. This is something that should be addressed in follow-up work on this.

Finally, Chapter 9 summarises some concluding observations, and suggests some possible routes for further work.

2 Evaluation framework

The tools were evaluated against the criteria that are defined in "WP 9: evaluation framework for characterisation tools". The full list of criteria is:

2.1 Tool interface

This relates to how a tool can be invoked. Examples are: command-line interface, Java API, SOAP. It is mainly relevant to the usability within a distributed architecture (e.g. tools that can only be invoked through a graphical user interface cannot be incorporated in an automated workflow without modification).

2.2 License type

If a tool is available under an open source license, it may be adapted or optimised to a distributed architecture. For closed-source tools this isn't usually an option (unless some agreement can be set up with the license holder).

2.3 Language

Open source tools may need to be adapted or optimised to the SCAPE architecture. However, if a tool is written in a language for which none of the involved SCAPE partners have sufficient experience, this may not be a viable option.

2.4 Platform dependencies

Some tools are only available for specific platforms (e.g. Windows XP). Such platform dependencies may make it difficult or even impossible to use such tools in other (e.g. Linux-based) environments.

2.5 Coverage of file formats

The selection of tools should take into account the file formats in which we are really interested (i.e. the formats that are prevalent in the collections of the SCAPE partners).

2.6 Extendibility

This relates to the degree to which it is possible to extend a tool to include new file formats, or update or improve the characterisation of existing formats. An example is DROID, for which the coverage of file formats can be extended by adding new format entries to an external signature file.

2.7 Output format

The incorporation into an automated workflow requires that tools report their output in a format that can be easily interpreted. XML formats that are properly defined by a schema are preferable.

2.8 Unique output identifiers

Tools may be using identifiers or tags that are different from the ones used in the format registry. An example: *BitsPerComponent* and *BitsPerSample* both refer to the same property of a JP2 image. The first is reported by ExifTool, the second by JHOVE1. This means that the output of individual tools will have to be normalised by mapping the properties to unique property instances in many cases.

2.9 Granularity of output

In addition to this, different tools may be using different levels of granularity. For instance, DROID reports its identification results as PUIDs, whereas the Unix File command uses MIME types instead.

As an example, any PDF file (irrespective of version) will return MIME type '/application/pdf' with the Unix File command, whereas DROID's PUID classification uses a separate PUID for each PDF version (and PDF profile). The tool evaluation should therefore take into account the required granularity, and the degree to which a tool's output can be mapped back to the used file format registry.

2.10 Accuracy of reported results

The information that is provided by a tool may not always be accurate. An example: ImageMagick's 'identify' tool reports the presence of embedded ICC profiles in JP2 images that do not contain an ICC profile at all.

2.11 Comprehensiveness and completeness of reported results

Candidate tools should also be evaluated for the completeness of the reported results. Another JPEG 2000 example to illustrate this: if a JPEG 2000 (JP2 or JPX) image contains an embedded ICC profile, JHOVE1 will report the used colour specification method ('Restricted ICC' or 'Any ICC'), which is important for preservation. However, it does not provide any information on the ICC profile itself. ExifTool on the other hand gives some very detailed output on the ICC profile, but it doesn't give any information on the colour specification method. So tools may be complementary to each other.

2.12 Fit to needs of preservation community

Many characterisation tools were not originally developed with the digital preservation community in mind. An example is ExifTool, which is more aimed at photographers. This is not necessarily a problem, but the evaluation should address to what extent tools meet the specific needs for preservation.

2.13 Ability to deal with nested objects

Not all tools are able to deal with nested objects. With the rise of formats such as EPUB and the Open Document Format, which all use the ZIP format as a physical container, dealing with such objects in a meaningful way is getting increasingly important.

2.14 Ability to deal with composite objects

Not all tools are able to deal with composite objects. Examples are (again) EPUB and the Open Document Format (where the constitute parts are held together by a physical ZIP container), and HTML.

2.15 User documentation

Tools that are used in an operational workflow should be supported by user documentation of sufficient quality. This should cover all aspects that are relevant to using the tool: system requirements, dependencies, limitations, installation, use (including options), and the interpretation of its output.

2.16 Computational performance

The computational performance of the tools should meet some minimal requirements. These are mainly related to speed and memory usage. If a tool is very slow, this may make its deployment in a large-scale automated workflow problematic. The same applies to a tool that consumes excessive amounts of system resources. For practical reasons, it is useful to make a distinction between the following two use cases:

1. Performance while working on 1 object at a time
2. Performance while working on a (very) large number of objects.

The reason for this is that many tools (especially the ones that are written in Java) suffer from slow initialisation after their invocation (due to slow startup of the Java Virtual Machine and the need to load large signature files). However, if such tools are able to perform many characterisations in one run, the average performance per object may still be very good. However, this may imply some specific requirements for the SCAPE framework. Conversely, in some cases the performance of tools that are designed to process only one object at a time may be improved by enabling the processing of many objects in one run.

2.17 Stability

The stability of the tools should meet some minimal requirements (also related to error handling and development stage, see below).

2.18 Error handling and reporting

The handling and reporting of errors should meet some minimal requirements. Any input/output errors and unexpected exceptions should not lead to system crashes, but they should be handled by the tool. The tool should also be able to report back any errors in such a way that they can be handled properly by the workflow management system.

2.19 Provision of event information

In addition to error handling and reporting, tools should ideally also provide information about each individual characterisation event, such as:

- Version number of the tool
- Version number of the signature file (in case of identification tools that are based on external signature files)
- Outcome: did the tool succeed, fail, or something in between? For identification tools this could include information on whether a tool was able to come up with a positive match for a specific file format, a tentative one, or no match at all.

2.20 Maturity and development stage

We may have reason to have more confidence in tools that have already seen a number of stable releases than in tools that are still in alpha or beta status. However, there are examples to the contrary, and the very nature of SCAPE as a research project implies that we shouldn’t overlook ‘new’ tools that show promise.

2.21 Development activity

Attempts to include or modify tools that show no signs of recent active development may turn out to be a waste of time and effort in the long run (lack of support, updates and user base). (There may be exceptions where SCAPE could potentially act as a trigger to re-activate dormant development efforts; the Open Planets Foundation could play an important role here.)

2.22 Existing experience

If we already have some experience with a tool, and are familiar with its use, chances are we are already aware of its strengths, weaknesses and limitations.

3 Data set and test environment

3.1 KB Scientific Journals set

This is a set of mostly scientific publications that were extracted from the KB's e-Depot repository. The data set comprises 7796 archival packages, containing a total of 11892 file objects (including both content and associated metadata files). Its total (uncompressed) size is about 1.15 GB. The materials in the data set originate from a wide variety of publishers. It has the following general structure, where each AIP is represented by a directory, which results in the following directory tree:

```
+---prdrm_1289223594188
|   \---unpacked
|       +---Original metadata
|       \---OriginalEpublication
|
+---prdrm_1289223611485
|   \---unpacked
|       +---Original metadata
|       \---OriginalEpublication
|
+--- etc.
```

The above example shows 2 AIPs. For each, the 'OriginalEpublication' directory contains the content files, and metadata are stored in the 'Original metadata' directory. Table 3-1 and Figure 3-1 give an overview of the data set's file size distribution. The Figure shows a marked peak around file of about 1 KB; this peak is mostly made up of small metadata files that are part of the archival packages. In general, large files appear to be somewhat underrepresented. For this reason, for some of the tests in this report an additional data set of (very) large files was used.

Table 3-1 Summary file size statistics of KB Scientific Journals data set (expressed in bytes). N=number of files; q1, median and q3 are 1st , 2nd and 3rd quantiles, respectively.

N	min	q1	median	mean	q3	max
11,892	11	955	4,737	104,650	43,177	25,495,289

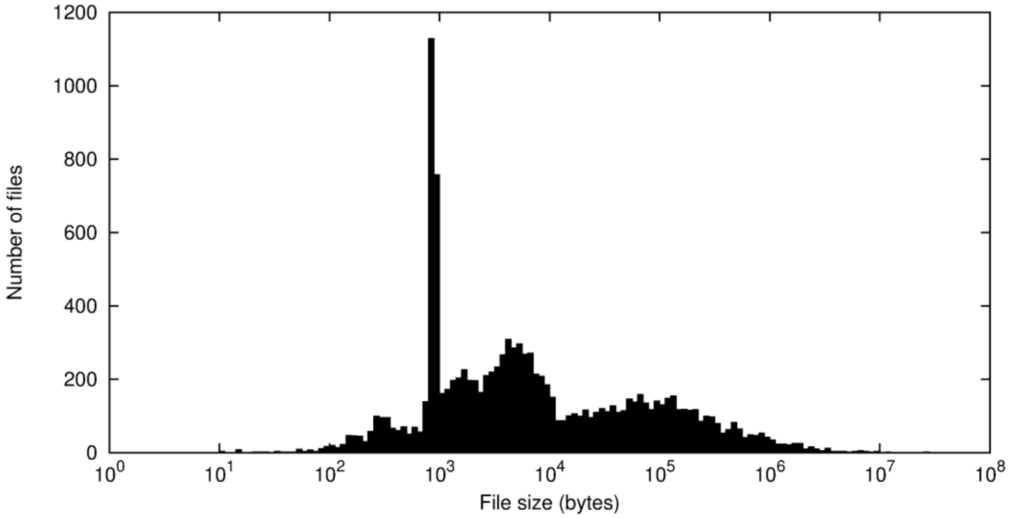

Figure 3-1 Distribution of file size in KB Scientific Journals data set. Note logarithmic scale on horizontal axis.

3.2 KB Large set

This is a data set that consists of 8 large files. Table 3-2 below lists its main characteristics.

Table 3-2 Main characteristics of KB Large data set

File name	File type	Size (bytes)
dpo_tonal_00990.tiff	TIFF image	36,420,900
IMAGE000060_lossless_colour.jp2	JP2 image	49,818,414
IMAGE000060.TIF	TIFF image	114,664,502
SGD_19451955_0000002_ID371.pdf	PDF 1.4	325,101,508
DipAsset6984444754559678047.tar	TAR archive	534,968,320
KBDVD.iso	ISO image	683,180,032
KBDVD17062011.img	ISO image	693,993,472
DNEPABOstg.PST	MS Outlook email folder	1,895,515,136

3.3 Test environment

All tests were run on a single desktop PC running Microsoft Windows XP Professional. The test dataset was stored on an external USB drive. Table 3-3 below summarises the most important system characteristics.

Table 3-3 Main characteristics of test environment

Processor	x86 Family 15 Model 6 Stepping 5 GenuineIntel ~2992 Mhz (32 bits)
Total Physical Memory	999 MB
Maximum virtual memory	2,048 MB
Operating System	Microsoft Windows XP Professional
OS Version	5.1.2600 Service Pack 3 Build 2600
Hard drive	LaCie Grand 1 TB USB 2.0 External
Java Runtime Environment	java version "1.6.0_26" Java(TM) SE Runtime Environment (build 1.6.0_26-b03) Java HotSpot(TM) Client VM (build 20.1-b02, mixed mode, sharing)

For the performance tests, two tools were used for measuring processing time:

1. For single tool invocations (e.g. processing of one single file, or processing of whole directory tree using a single invocation of a tool) we used Microsoft's 'timeit' utility. It is part of the Windows Server 2003 Resource Kit Tools[2].
2. Some of the tests involved a recursive traversal of the whole directory tree, where all 11892 file objects were processed using 11892 separate tool invocations. This was automated with a custom-made Python script called 'treeLaunch', which runs a user-defined command as a sub-process on all file objects in a directory tree. The script includes a built-in timer which measures the amount of time between the start and completion of each sub-process. For each file/sub-process, the script reports the file path of the corresponding file object, its MIME type[3], its size and the duration of the subprocess.

Manual tests on a limited number of tool / file combinations revealed no noticeable differences between the results of both tools.

[2] More info here: http://www.microsoft.com/downloads/en/details.aspx?FamilyID=9d467a69-57ff-4ae7-96ee-b18c4790cffd&displaylang=en

[3] MIME type here is guessed based on file extension only Python's built-in 'mimetypes' module

4 DROID 6.0

4.1 Overview

DROID (Digital Record Object Identification) is a tool that identifies digital objects using PRONOM format signatures ('magic numbers') and/or known file extensions. The identification results are reported as PRONOM-compliant Persistent Unique Identifiers (PUIDs). DROID is an open-source, platform-independent Java application. It can be used directly from the command line, or, alternatively, using a graphical user interface. Evaluated here is the most recent version of DROID , which is DROID 6.0 (released March 2011).

4.2 Tool interface

DROID can be invoked through a graphical user interface and a command line interface. For the command line interface a Windows batch file and a Unix shell script are provided. The batch file only appears to work if it is invoked directly from the application directory (i.e. the directory in which the batch file and the two main JAR files are installed). Since this is not very practical, calling the executable JAR of the command-line interface directly[4] will often be more convenient. It is possible to tell DROID to analyse all files in a given directory (or set of directories), and the "-R " command-line switch activates recursive processing of all subdirectories (and sub-sub directories, and so on).

4.3 License type

DROID is released under an open-source BSD License, which "permits its use, modification, inclusion in other products and redistribution as long as the terms of the license are adhered to" [DROID 6]. DROID uses a number of third party components, which are all released under a variety of open source license types.

4.4 Language

DROID is written in Java (version 6). The user documentation states that DROID has been tested on Java 6 update 17 to update 22, while it should run under Java 6 update 10 onwards.

4.5 Platform dependencies

According to the user documentation DROID runs on Windows, Linux and Mac, and potentially other platforms that support Java 6 applications (see above).

4.6 Coverage of file formats

DROID tries to identify files using file signatures, or, alternatively, known extensions. These are defined in a signature file, which is regularly updated by The National Archives. Just as an indication, version 45 of the signature file contains 766 file type definitions and 327 file signatures.

4.7 Extendibility

Since the number of formats that DROID can handle is defined by the information in the signature file, new formats can be added by modifying the signature file (or downloading the latest version from the National Archives). Unlike earlier versions, in DROID 6 the signature file is always stored in a separate (configurable) user directory (which is not necessarily the location of the DROID application files). In addition to the regular signature file, DROID 6 also uses a separate file with 'container signatures ', which is used for formats that may contain other files within themselves. Examples are

[4] E.g. like this: `java -jar C:\droid\droid-command-line-6.0.jar -a ...`

ZIP, which is used as a physical container for the Open Document and Microsoft Office Open XML formats, and OLE2 which is a container for the Microsoft Office formats.

4.8 Output format

Older versions of DROID (e.g. DROID 4) used to write their output directly to an XML file. In DROID 6 things are slightly more complicated. Central to its input and output handling is the concept of *profiles*. A profile is defined as "the files and folders you want to find out about, and the results of profiling them". A profile uses the ZIP format as a physical container, and all relevant DROID in- and output results are stored in this container. Crucially, the characterisation results are stored in a database format that is not human readable. In order to use the results, the information from the profile has to be exported. DROID 6 can only export the profile information to CSV (comma-separated text) format. By default, each row in the exported CSV file represents one file object. However, if a file object results in multiple format matches, the last 4 columns of the CSV output (PUID, MIME type, format name and format version) are repeated for each match. This means that the number of columns per row is variable. Alternatively, DROID offers a 'one row per format' option, where each row represents one unique format identification (which has the implication that one file objected may be represented by multiple rows in the CSV file).

In addition to this, DROID 6 also has options to generate *reports* (which can be in either XML or PDF format), but this report functionality is mainly useful for generating aggregate statistics of the identification results (e.g. for each format the corresponding number of files). None of these output formats are particularly suitable for use in automatic workflow systems[5]. Exporting is done as a separate step, which means that 2 separate invocations of DROID are necessary to produce the CSV output (i.e. first scan, then export).

4.9 Unique output identifiers

Identification results are reported as PRONOM Unique Identifiers (PUID) and MIME types. In addition, DROID also provides a textual description of the format name and (if applicable) the format version number.

4.10 Granularity of output

The use of PUID as the primary identifier ensures that there DROID's output can be mapped directly to the PRONOM registry (as well as the OPF registry).

4.11 Accuracy of reported results

Not analysed yet.

4.12 Comprehensiveness and completeness of reported results

The reporting of the analysis results appears to be complete and sufficiently comprehensive, with the important exception of event information, which is explained further below. The output also contains the full URI and file path for each analysed object.

[5] We have contacted The National Archives about these output format issues, and inquired whether it would be possible to re-introduce the 'old' XML output format. In a reply to this (e-mail Andrew Fetherston, 11-4-2011) TNA confirmed that the XML output was dropped in recent DROID releases. He did however offer to add a suggestion for unified XML output of results for any future development of DROID.

4.13 Fit to needs of preservation community

DROID is being developed by The National Archives, and the preservation community is its primary target audience. However, the encountered output-related problems suggest that the current focus of DROID is on interactive use of the software using the Graphical User Interface. The needs of users who wish to incorporate DROID 6 in automated workflows are not served particularly well. The inclusion of more elaborate XML output (similar to what was used in e.g. DROID 4) would make the software much more suitable for such applications.

4.14 Ability to deal with nested objects

DROID 6 is able to look inside ZIP files. DROID analyses all file objects that are embedded in a ZIP file, and these are included in DROID's output. To illustrate this, the example below shows how a Perl script that is embedded inside a ZIP archive is reported:

```
zip:file:/D:/aipSamplesUnpacked/./prdrm_1289288371730/unpacked/Original%20m
etadata/1746-4811-1-10-S2.zip!/CDF_masking/easy_script.pl
```

4.15 Ability to deal with composite objects

Formats such as Microsoft Word 97 and Open Document Format are based on multiple file objects that are held together by a physical container (e.g. OLE2 for Microsoft Word 97, and ZIP for Open Document Format). DROID 6 can handle both cases. Open Document files are identified as Open Document format (and not as plain ZIP, even though ZIP is the container format). Interestingly, EPUB files (which are organised in a similar manner as Open Document files) are still treated (and identified) as regular ZIP files.

HTML represents another class of composite objects, where, for example, an individual HTML file refers to external style sheets and images, which are all needed for proper rendering. The main difference with e.g. Open Document Format or EPUB is the absence of a physical container file. DROID does not currently have any mechanism to recognise the interdependencies between the individual components of this particular class of 'composite objects'.

4.16 User documentation

The GUI version of DROID has a comprehensive on-line help. Its contents are also provided as a 65 page manual in PDF format. The current evaluation revealed the following issues:

- In the description of the CSV output some columns are left out, and the column headings in the documentation are sometimes slightly different from those in the actual CSV files (see also the section on event information below).
- The documentation of the CSV output doesn't explain that additional columns are added in case of multiple format matches.
- The manual describes how to configure folders for user settings and temporary files. The actual behaviour of the temporary files directory is slightly different from the documented behaviour. In particular, by default the temporary files directory settings should be inherited from the user settings directory settings, but this hasn't been implemented accordingly in the software. The result is that DROID may end up putting large amounts of temporary data at unexpected locations, which can lead to system problems[6].

[6] This issue was reported to TNA, and they confirmed that this is a bug that will be corrected in upcoming releases. In the meantime a simple fix is to always set the temporary files folder explicitly using the 'droidTempDir' environment variable (as described on page 62 of the DROID 6 Help document).

- In order to be able to work with DROID 6 a basic understanding of the 'Profile' concept is crucial. However, the documentation does not clearly explain *what* a 'Profile' exactly *is*. The most specific description describes a profile as "the files and folders you want to find out about, and the results of profiling them", but apart from being rather vague it is hidden away in the 'Create a new profile' section.
- The PDF documentation is a direct derivative of the on-line help. Although certainly very useful, it is somewhat lacking in structure (see also the aforementioned remark on profiles). A properly edited user manual (which would, for example, explain central concepts such as 'profiles' in an introductory chapter) could be a further improvement.

4.17 Computational performance: one file at a time

4.17.1 KB Scientific Journals data set

DROID's performance while processing one file at a time was tested using the 'treeLaunch' tool (see section 3.3). The 'treeLaunch' tool was set up to recursively traverse the KB Scientific Journals data set's directory tree, and run DROID for each encountered file object. For an individual file object this results in a command line like:

```
java -jar C:\droid\droid-command-line-6.0.jar -a 1-1-70.pdf -p test1.droid
```

Here, DROID's *–a* switch defines a resource (in this case a PDF file), and *–p* defines a 'profile' (see above).

It took about 39 hours to analyse the KB Scientific Journals data set in this way. Figure 4-1 and Table 4-1 summarise the main results. On average DROID 6 needs about 12 seconds per file object, but in exceptional cases more than 20 seconds are needed. Figure 4-2 shows that processing time is independent of file size.

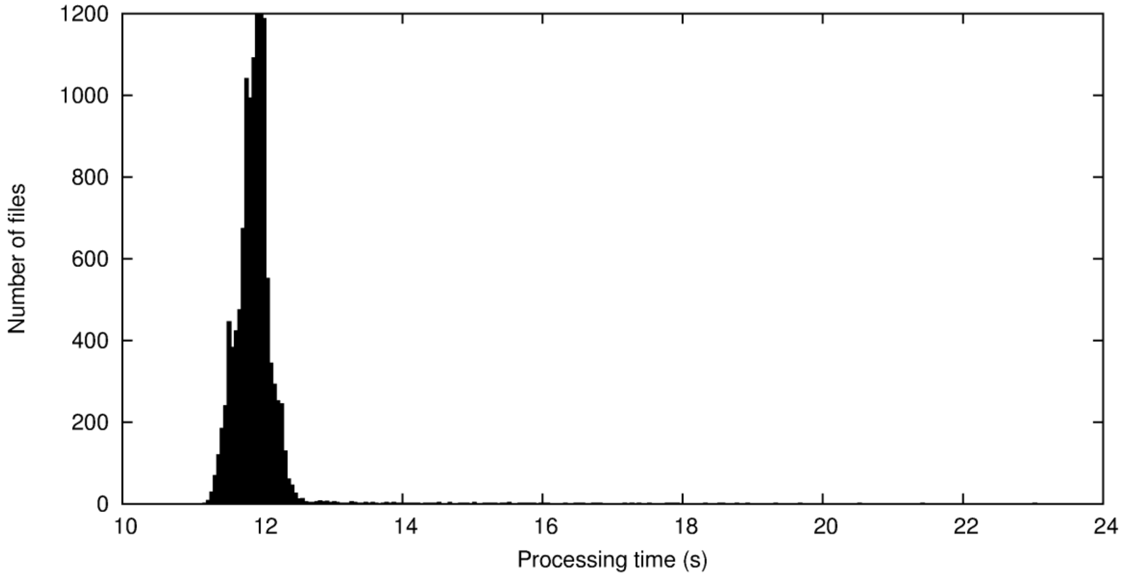

Figure 4-1 Distribution of processing time per file object for DROID 6 one-file-at-a-time scenario, KB Scientific Journals data set.

Table 4-1 Summary performance statistics for DROID 6 one-file-at-a-time scenario, KB Scientific Journals data set (expressed in seconds per file, except N). N=number of files; q1, median and q3 are 1st , 2nd and 3rd quantiles, respectively.

N	min	q1	median	mean	q3	max
11,892	11.19	11.74	11.89	11.90	12.00	23.02

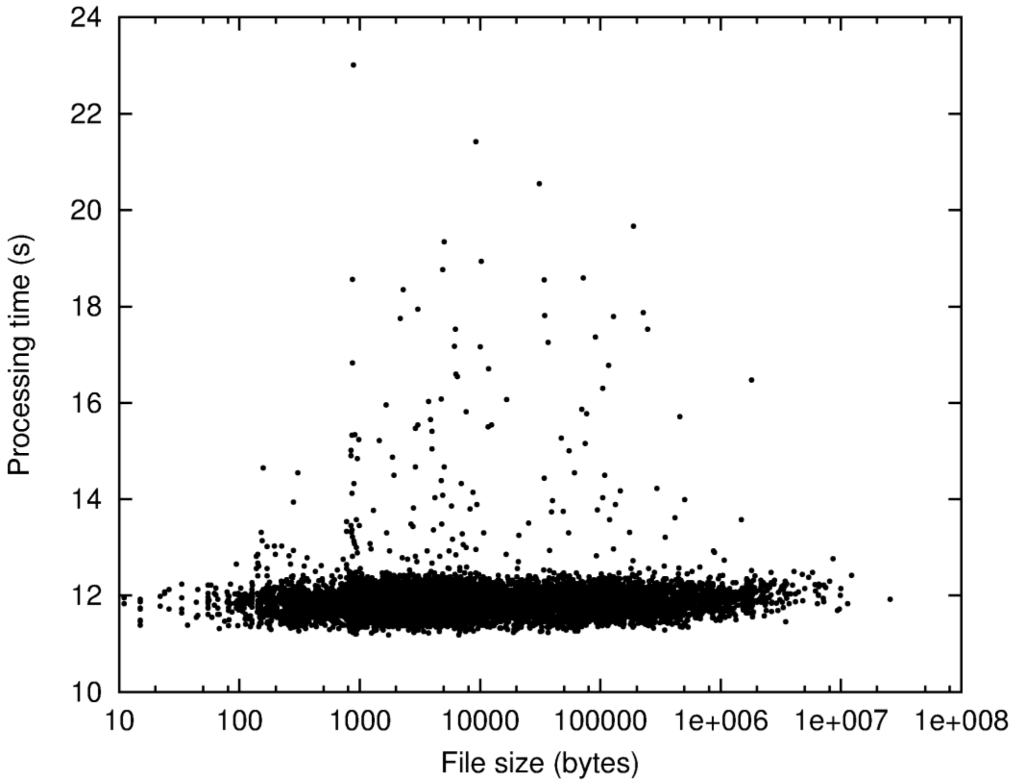

Figure 4-2 Scatter plot of processing time per file object versus file size for DROID 6 one-file-at-a-time scenario, KB Scientific Journals data set. Note logarithmic scale on horizontal axis.

4.17.2 KB Large data set

Table 4-2 shows the results of an additional test that was done on the KB Large data set (again using the 'treeLaunch' tool). The table demonstrates that in general file size does not appear to affect processing time very much. The TAR archive is an exception; however, bearing in mind that TAR is a container format for which DROID also analyses all packed file objects this is not surprising.

Table 4-2 Performance for individual files in KB Large data set

File name	Size (bytes)	Processing time (s)
dpo_tonal_00990.tiff	36,420,900	12.63
IMAGE000060_lossless_colour.jp2	49,818,414	12.38
IMAGE000060.TIF	114,664,502	12.53
SGD_19451955_0000002_ID371.pdf	325,101,508	19.80
DipAsset6984444754559678047.tar	534,968,320	65.28
KBDVD.iso	683,180,032	12.53
KBDVD17062011.img	693,993,472	13.52
DNEPABOstg.PST	1,895,515,136	15.88

4.18 Computational performance: many files at a time

To test DROID’s performance while working on a large number of objects, we performed a recursive scan of all file objects in the directory structure of the KB Scientific Journals data set and measured the time needed for this. We used the following command line:

```
timeit java -jar C:\droid\droid-command-line-6.0.jar -a . -R -p
resultsAll.droid
```

Here, DROID’s –*a* switch again defines a resource (in this case the current directory); -*R* activates recursing into subdirectories, and –*p* defines a ‘profile’ (see above).

This gives the following result:

Elapsed Time:	0:03:12.390

So DROID 6 needs about 3 minutes to scan 1.15 GB of data in the used test environment. To get human- or machine-readable output, the resulting profile needs to be exported to CSV format, which requires an additional DROID invocation:

```
timeit java -jar C:\droid\droid-command-line-6.0.jar -p resultsAll.droid -e
resultsAll.csv
```

The timing result in this case:

Elapsed Time:	0:00:22.890

At 20 s for about 20 thousand records (almost 12 thousand file objects + about 8 thousand directories) this means that the additional overhead by the exporting step is unlikely to be a problem.

These (preliminary) tests suggest that the computational performance of DROID 6 is rather good, provided that a sufficiently large number of objects is analysed during each DROID run

4.19 Stability

Not investigated in detail so far. However, the current tests did not result in any crashes or other stability issues.

4.20 Error handling and reporting

The DROID manual states that errors are sent to the console’s standard error output.

4.21 Provision of event information

The XML output format of older DROID versions used to contain event information on the DROID version, the signature version, and the identification status of each file. Most of this has been abandoned in the CSV output of DROID 6. More specifically:

- The DROID version number is not included
- The version number of the signature file is not included
- Event information on the outcome of individual analysed files is largely limited to system errors only (e.g. errors that arise because a file doesn’t exist or access to a file is denied). This

is recorded in the 'STATUS' column (which is somewhat confusingly called 'Job status' in the documentation).

- DROID 6 does provide information on the method that was used to identify an object (signature or extension), and any mismatches between extensions and signatures are recorded in an 'EXTENSION_MISMATCH' field (which is not described in DROID's documentation!).
- The output provides *no* information at all on whether an object could be identified in the first place: if DROID encounters an unidentifiable object, most of the output fields are simply left blank.

Summarising, the provision of event information (or lack thereof) appears to be a major problem when using DROID 6 in an automated workflow[7].

4.22 Maturity and development stage

The first version of DROID was released in 2006, and DROID 6 is the sixth major release. All major DROID versions are released as stable versions (so no alpha or beta releases).

4.23 Development activity

With six major releases in six years, the development of DROID appears to be very active. In addition to this, updated versions of the DROID signature file are released on a regular basis (e.g. at the time of writing the current signature file is version 49).

4.24 Existing experience

It appears that many memory institutions that are working with digital records have at least some experience with DROID, although it is not clear how many of them are really using the software operationally.

4.25 Unidentified files

Although not part of the evaluation framework, it is useful to know something about which files can and cannot be identified by a tool. DROID was unable to identify 566 files in the test dataset (some of these are objects that are nested inside a ZIP archive). The following table shows the file extensions that correspond to non-identified files:

Extension	Count	Remarks
fil	2	Text files with checksum values (Elsevier)
mol	3	MDL MOL file (chemical table format, http://en.wikipedia.org/wiki/MDL_Molfile)
oa3	69	XML metadata format
pl	3	Perl script
pm	4	Perl script
r	1	R script (R, programming language for statistical applications)
sgc	52	SGML metadata (Elsevier)

[7] We reported some of these problems back to the TNA. In a reply (e-mail Andrew Fetherston, 11-4-2011), they suggested to use the XML reporting feature (which is meant for generating automated reports with aggregate statistics on DROID runs) to generate some of the missing DROID and signature version output, and then combine the results of the CSV file and the XML report using some custom-written script. However, this looks like a step backward compared to DROID's previous XML output format.

sgm	199	SGML metadata (Elsevier)
toc	59	Text table specific to Elsevier
xml	172	XML files without XML declaration

Note that these are all text-based formats, some of which contain markup (SGML or XML). Also, DROID 6 cannot properly identify XML files that do not contain an XML declaration (which is not mandatory in XML). This is simply a limitation of identification based on file signatures: without an XML declaration, analysing the whole file with an XML parser would be the only way to establish its contents as XML.

4.26 Conclusions

The evaluation shows that DROID 6 is potentially suitable for inclusion in the SCAPE architecture. Since DROID 6 initialises quite slowly, it is important that it is deployed in such a way that many files are analysed in one run. Under this condition, DROID's computational performance appears to be more than adequate. The main foreseen problems are related to DROID's current output reporting options, which are not well suited to automated workflows. This applies in particular to the provision of identification results in CSV format (rather than XML), the output handling in case of multiple format matches (or no matches at all), and the lack of event information. Finally, the description of the CSV output files in DROID's documentation is incomplete.

5 Fido 0.9

5.1 Overview

Fido (Format Identification for Digital Objects) is an identification tool that also uses the PRONOM format signatures. It is essentially a DROID clone. The identification results are reported as PRONOM-compliant Persistent Unique Identifiers (PUIDs). Fido is an open-source command line application written in Python. There is also a Jython-based version of Fido (Jython is a Java implementation of the Python language), which provides Fido as a JAR file. The following two versions will be evaluated in this chapter:

1. Fido 0.9.3 (released December 2010)
2. Fido_jar 0.9.5 (Jython version, released March 2011)

Note here that no 'regular' 0.9.5 version of Fido has been released so far, which is due to the management of the code being transferred from the original developer (Adam Farquhar) to the Open Planets Foundation (Andy Jackson, personal communication). Since version 0.9.5 is the first (and currently only) Jython release, it was not possible to use the same version of both releases.

5.2 Tool interface

5.2.1 Python version

Fido can be invoked through a command line interface. On top of the Python script, a Windows batch file and a Unix shell script are provided. However, these are not very helpful since they contain a reference to a non-existing file (fido.run). Moreover, they are not really needed, as installing Python under Windows will automatically associate *.py* files with the Python interpreter, whereas on Linux-based systems the command-line interpreter can be explicitly declared on the first line of the script.

5.2.2 Jython version

Fido can be invoked through a command line interface.

5.3 License type

Fido is open-source software that is released under Apache version 2.0 license.

5.4 Language

Fido is written in Python. Currently it uses a Python syntax that is compatible with Python versions 2.6 and 2.7[8]. It is not compatible with the Python 3.x range of interpreters. Even though the 2.x range of Python interpreters is still widely used, it may be worthwhile to consider upgrading to Python 3.x – compatible code in order to make it more future-proof. In many cases it is possible to produce code that works under both Python 2.6 (and 2.7) and Python 3.x[9].

5.5 Platform dependencies

5.5.1 Python version

Fido runs on any platform for which Python 2.6 or Python 2.7 is available (it is currently not compatible with the Python 3.x range of interpreters). There are no other dependencies.

[8] Just one example is the use of the 'print' *statement* [example: print "hello world"], which is replaced by the 'print' *function* [example: print("hello world")] in Python 3.0 and later.

[9] In fact Python 2.6 was especially released as a transition version that already supports most of 3.x syntax.

5.5.2 Jython version

According to its documentation, the Jython version of Fido requires Java 6 Update 23 or later with no other dependencies.

5.6 Coverage of file formats

Fido tries to identify files using file signatures, or, alternatively, known extensions. These are defined in two files, which are based on information from the PRONOM database. So in principle all formats of the PRONOM database are covered, provided that they have a valid signature. Importantly, the format of these files is **not** identical to the DROID signature files.

5.7 Extendibility

As with DROID, new formats can be added by modifying or updating the signature and extension files. Although Fido's documentation describes how to add new formats to these files, it does not describe how to convert the PRONOM/DROID information to Fido-compatible format. Not having such an option would severely limit Fido's use in any operational setting[10].

5.8 Output format

Output is written to the standard output device, which can be redirected to a text file. The format is largely configurable using two user-defined command line parameters (matchprintf and nomatchprintf) which are formatting strings (in Python format) that define the output in case of a match or no match respectively. The default values of these parameters result in output in comma-delimited format, where each line represents one unique format hit (if one file object gives two format hits, they are both written separately). In principle the format strings could be used to generate XML tags, but the creation of well-formed XML would require the possibility to write some leading and trailing text data that could be used to define the root element at the start and the end of the output. In the simplest case this could be implemented through the use of 2 additional formatting strings.

5.9 Unique output identifiers

Identification results are reported as PRONOM Unique Identifiers (PUID) and MIME types[11]. In addition, Fido also provides a textual description of the format name and (if applicable) a signature name.

5.10 Granularity of output

The use of PUID as the primary identifier ensures that there Fido's output can be mapped directly to the PRONOM registry (as well as the OPF registry).

5.11 Accuracy of reported results

Not analysed yet.

[10] In a response to an earlier version of this report, Maurice de Rooij (OPF / NANETH) comments that this conversion can be done with the (undocumented) 'prepare.py' script that is part of Fido. A first test of this script wasn't successful, but we will look into this later.

[11] However, MIME type reporting doesn't seem to work the way it should, see Section 5.2.11

5.12 Comprehensiveness and completeness of reported results

Overall, Fido's output is less verbose than DROID. Fido can be configured to report any or all of the following output variables (see Fido's documentation for more information):

- count (nth item matched)
- group_size
- filename
- filesize
- time (in msecs)
- group_index
- puid,
- formatname
- signaturename
- mimetype

This does not include the identification status (i.e. could a file object be identified at all), but this can be derived indirectly through the use of the 'matchprintf' and 'nomatchprintf' strings. A cursory inspection of some test results showed that if relative file paths are used for the FILE argument, file names are written to the output using relative paths as well. This could be a problem in some cases, whereas it may be desired in others (if full paths are used on the command line FIDO will report absolute paths; if relative paths are given on the command line relative paths are reported). In addition, some quick tests with the '-matchprintf' option revealed that reporting of MIME types always produces a 'None' result, which doesn't look quite right. As with DROID, the possibilities for providing event information are rather limited.

5.13 Fit to needs of preservation community

Fido is being developed under the umbrella of the Open Planets Foundation, and the preservation community is its primary target audience. Unlike DROID, the focus appears to be on use in automated workflows.

5.14 Ability to deal with nested objects

5.14.1 Python version

Fido is able to look inside ZIP files. It analyses all file objects that are embedded in a ZIP file, and these are included in the output. To illustrate this, the example below shows how a CSV file that is embedded inside a ZIP archive is reported:

```
".\prdrm_1289288371730\unpacked\Original metadata\1746-4811-1-10-
S1.zip!Supplementary Table I.csv"
```

5.14.2 Jython version

The Jython version of doesn't yet support ZIP recursing.

5.15 Ability to deal with composite objects

Formats such as Microsoft Word 97 and Open Document Format are based on multiple file objects that are held together by a physical container (e.g. OLE2 for Microsoft Word 97, and ZIP for Open Document Format). Microsoft Office formats (Word, Excel) are all identified as "OLE2 Compound Document Format", which means Fido only recognizes the container, which is not very helpful. Open

Document files are identified as Open Document format (and not as plain ZIP, even though ZIP is the container format). EPUB files (which are organised in a similar manner as Open Document files) are still treated (and identified) as regular ZIP files.

HTML represents another class of composite objects, where, for example, an individual HTML file refers to external style sheets and images, which are all needed for proper rendering. The main difference with e.g. Open Document Format or EPUB is the absence of a physical container file. Like DROID, Fido does not currently have any mechanism to recognise the interdependencies between the individual components of this particular class of 'composite objects'.

5.16 User documentation

5.16.1 Python version

The documentation of Fido is limited to a readme text file, which explains the command line interface, the installation of Fido, and its generated output. The evaluation revealed the following problems:

- The installation instructions describe the use of an installation script. Running this script under Windows results in the installation of Fido in a number of (subdirectories of) the Python system directories. However, this is not described in the documentation (which doesn't even mention the installation location at all).
- For running Fido, the documentation erroneously refers to a file called "fido.run", which doesn't exist (this should probably be "run.py").
- To add to the confusion, in the readme information on the main page of Fido's Github site this file is simply called "fido.py", which probably reflects the -yet unreleased- 0.9.5 version (although the same document also refers to the non-existent "fido.run" file) .
- The method for calling Fido that is given in the documentation looks unnecessarily verbose. According to the readme, the following command line should be used:

  `python -m fido.run -h` (which should really be `python -m run.py -h`!)

  However, in most cases[12] this can be reduced to:

  ```
  run.py
  ```

  This also eliminates the need for using either the batch file or shell script that are provided with Fido (which aren't working anyway!).

5.16.2 Jython version

The Jython version comes with a readme text file that is similar to the one of the Python version. None of the comments on the Python version above apply to the Jython version, and the evaluation didn't reveal any particular problems.

[12] A possible exception would be a Windows-based system on which multiple Python versions are installed.

5.17 Computational performance: one file at a time (Python version)

5.17.1 KB Scientific Journals data set

The 'treeLaunch' tool was used to get an impression of Fido's performance when it is called to process one file at a time. It was set up to recursively traverse the KB Scientific Journals data set's directory tree, and run Fido for each encountered file object. For an individual file object this results in a command line like[13]:

```
c:\python27\python c:\fido\run.py -zip 1-1-70.pdf
```

Here, Fido's *–zip* switch activates recursing into ZIP files. The *–zip* switch was explicitly activated here in order to make the test results comparable with those of DROID (which recurses into ZIP files by default).

It took 1 hour and 6 minutes to analyse the whole KB Scientific Journals data set in this way. Figure 5-1 and Table 5-1 summarise the main results. On average Fido needs about 0.3 seconds per file object, with a maximum of about 1 second. This means that for the one-file-at-a-time scenario Fido's overall performance beats DROID 6 by a factor of about 36. Interestingly, Figure 5-2 suggests that processing time increases slightly with file size.

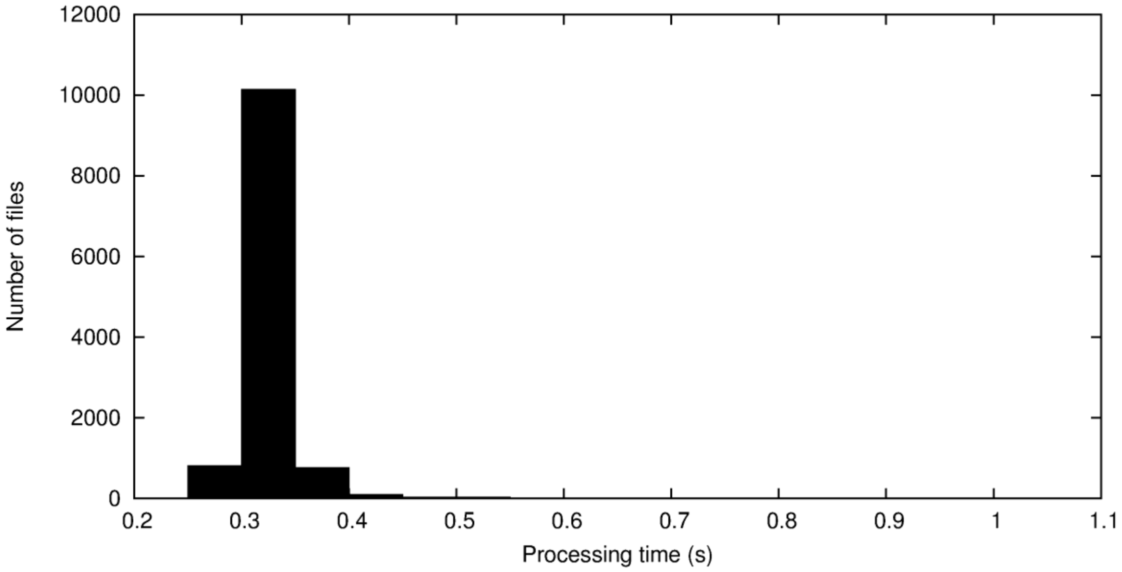

Figure 5-1 Distribution of processing time per file object for Fido one-file-at-a-time scenario, KB Scientific Journals data set.

Table 5-1 Summary performance statistics for Fido one-file-at-a-time scenario (expressed in seconds per file, except N). N=number of files; q1, median and q3 are 1st , 2nd and 3rd quantiles, respectively.

N	min	q1	median	mean	q3	max
11,892	0.2960	0.3120	0.3130	0.3219	0.3280	1.0780

[13] The explicit call to the Python interpreter was included here because omitting it resulted in a problem with the 'treeLaunch' tool. In most cases it can be omitted, and calling the Python script directly will work fine.

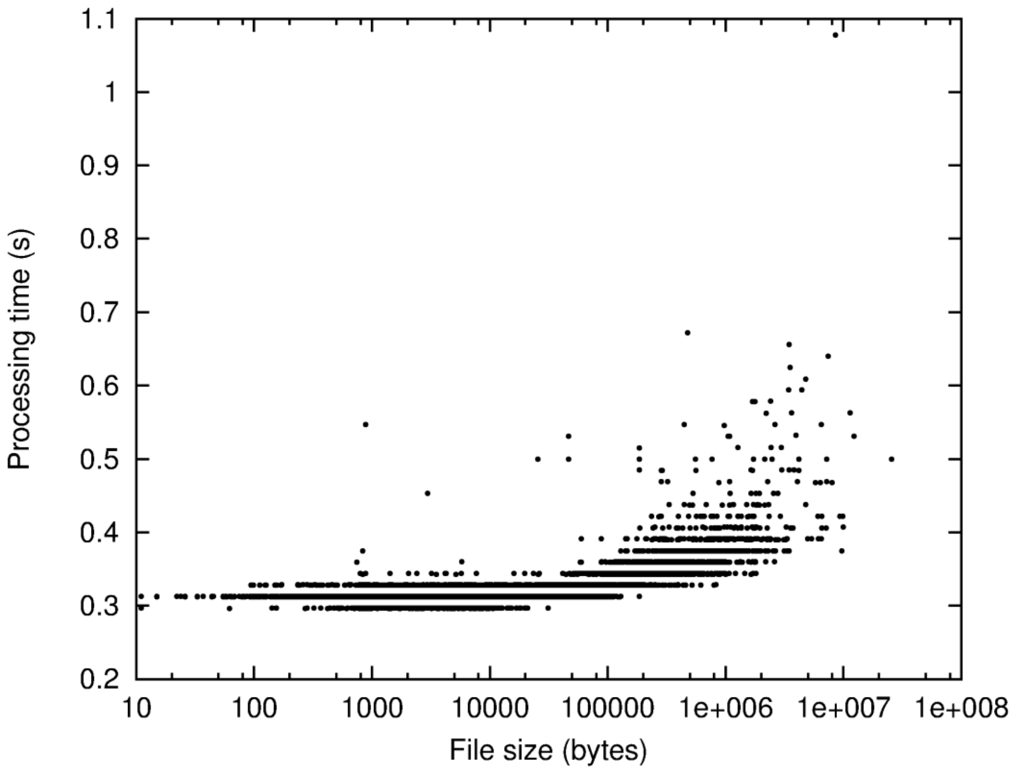

Figure 5-2 Scatter plot of processing time per file object versus file size for Fido one-file-at-a-time scenario, KB Scientific Journals data set. Note logarithmic scale on horizontal axis.

5.17.2 KB Large data set

Table 5-2 shows the results of an additional test that was done on the KB Large data set (again using the 'treeLaunch' tool). The table demonstrates that in general file size does not appear to affect processing time very much. Again the TAR archive is an exception; however, bearing in mind that TAR is a container format for which Fido also analyses all packed file objects this is not surprising.

Table 5-2 Performance for individual files in KB Large data set

File name	Size (bytes)	Processing time (s)
dpo_tonal_00990.tiff	36,420,900	0.375
IMAGE000060_lossless_colour.jp2	49,818,414	0.359
IMAGE000060.TIF	114,664,502	0.375
SGD_19451955_0000002_ID371.pdf	325,101,508	0.359
DipAsset6984444754559678047.tar	534,968,320	17.532
KBDVD.iso	683,180,032	0.36
KBDVD17062011.img	693,993,472	0.359
DNEPABOstg.PST	1,895,515,136	0.375

5.18 Computational performance: one file at a time (Jython version)

For the Jython version we started with the following command line. Since the documentation states that ZIP recursing doesn't yet work for the Jython version of Fido, the –zip option was not used in any of these tests.

```
timeit java -jar  C:\Temp\identtools\fido_jar-0.9.5\fido.jar 1-70.pdf >
jfidoOut.csv
```

However, although Fido didn't produce any error messages, this resulted in an empty output file. After some experimentation it turned out that the Jython version of Fido only works if it is called from its installation directory (in this case: C:\Temp\identtools\fido_jar-0.9.5\). This appears to be a bug. As a workaround we used this command line instead:

```
timeit java -jar fido.jar D:\aipSamplesUnpackedOneDir\1-70.pdf >
jfidoOut.csv
```

However, this time Fido exited with the message "FIDO: Processed 0 files in 13399.13 msec", with again an empty output file. So in a final attempt we replaced the backward slashes by forward ones:

```
timeit java -jar fido.jar D:/aipSamplesUnpackedOneDir/1-70.pdf >
jfidoOut.csv
```

This gave identical results to the ones presented above. So the preliminary conclusion is that the Jython version of Fido is not able to process files one at a time.

5.19 Computational performance: many files at a time

To test Fido's performance while working on a large number of objects, we performed a recursive scan of all file objects in the directory structure of the KB Scientific Journals data set and measured the time needed for this.

5.19.1 Python version

For the Python version initially used the following command line [14]:

```
timeit C:\Python27\python C:\Temp\identtools\fido-0.9.3\fido\run.py -
recurse -zip . > fidoOut.csv
```

Here, Fido's *-recurse* switch activates recursing into subdirectories, and *–zip* activates recursing into ZIP files (see above). Output is again redirected to a comma-delimited text file ('fidoOut.csv').

This gives the following result:

Elapsed Time:	0:03:35.764

This is similar to the corresponding DROID result, although Fido is marginally slower. However, unlike DROID, no subsequent exporting steps are needed to get human- or machine-readable output.

[14] The explicit call to the Python interpreter was included here because omitting it resulted in a problem with the 'timeit' utility. In most cases it can be omitted, and calling the Python script directly will work fine.

After reviewing the resulting output of Fido, it turned out that the number of files that could not be identified was significantly larger than was the case with DROID (1385 for Fido compared to 566 for DROID 6). Upon closer inspection, this is probably due to the fact that Fido's default behaviour is to identify solely based on byte signatures, whereas DROID also uses file extensions if it cannot find a match for any of the format signatures. Fido has an *–extension* command line switch that, when activated, should result in similar behaviour. However, using this option resulted in a run-time error ('AttributeError'). Because of this, the *–extension* switch was not used in these tests.

5.19.2 Jython version

For the Jython version we started with the following command line[15] (since the documentation states that ZIP recursing doesn't yet work for the Jython version of Fido, the –zip option was not used in any of these tests):

```
timeit java -jar  C:\Temp\identtools\fido_jar-0.9.5\fido.jar -recurse . >
jfidoOut.csv
```

However, although Fido didn't produce any error messages, this again resulted in an empty output file. After some experimentation it turned out that the Jython version of Fido only works if it is called from its installation directory (in this case: C:\Temp\identtools\fido_jar-0.9.5\). This appears to be a bug. As a workaround we used this command line instead:

```
timeit java -jar  fido.jar -recurse D:\ aipSamplesUnpacked > jfidoOut.csv
```

However, this time Fido exited with the message "FIDO: Processed 0 files in 13367.23 msec", with again an empty output file.

Next we added a trailing backslash:

```
timeit java -jar  fido.jar -recurse D:\aipSamplesUnpacked\ > jfidoOut.csv
```

This raised an exception in Jython ("SyntaxError: ("mismatched character")).

In a final attempt we replaced all backward slashes by forward ones:

```
timeit java -jar  fido.jar -recurse D:/aipSamplesUnpacked/ > jfidoOut.csv
```

This produced the very first run of the Jython version of Fido that resulted in any output. However, upon inspection of the output it turned out that in spite of having used the –recurse switch, Fido hadn't actually recursed into any of the subdirectories. This meant that only 6 (out of a total of 11892) file objects files were analysed! In a last attempt to get at least some idea of this tool's performance, Fido was run on a modified version of the dataset in which all file objects are in one single directory:

```
timeit java -jar fido.jar -recurse D:/aipSamplesUnpackedOneDir/ >
jfidoOut.csv
```

[15] As it happens, the *–extension* switch *does* actually work for the Jython version, but it was not used in any of these tests.

Giving the following result:

Elapsed Time:	0:49:05.387

So, where both DROID 6 and the Python version of Fido need about 3.5 minutes, the Jython version takes no less than 49 minutes. Compared to the Python version of Fido, the Jython version is roughly 14 times slower.

5.20 Stability

5.20.1 Python version

Although this has not been investigated in any detail so far, during the performance tests run-time errors occurred while using the –*extension* switch (see above).

5.20.2 Jython version

From the performance tests above it is obvious that stability is a major problem for the Jython version of Fido.

5.21 Error handling and reporting

Fido's documentation does not mention this, but from the tests it appears that error messages are written to the standard error device.

5.22 Provision of event information

Fido provides only a minimum amount of event information

- The Fido version number is not included
- The version number of the signature file is not included – in fact, Fido's signature and extension files do not appear to include any version info at all (which could lead to several unpleasant scenarios)
- Information on the method that was used to identify an object (signature or extension) can be derived indirectly from the 'signaturename' field in the output.
- Contrary to DROID, Fido *is* able to provide information whether an object could be identified in the first place (and this information is also reported by default).

Summarising, the provision of event information is in many ways similar to DROID 6, although being able to establish whether a file object could be identified at all is very useful. The current absence of any versioning information in the signature and extension files makes any version management in operational settings virtually impossible.

5.23 Maturity and development stage

The first version of Fido was released in November 2010, and its authors currently consider Fido to be in beta stage. The Jython version was released much more recently, and the outcome of the current tests suggest that a pre-alpha stage might be more appropriate for this particular version.

5.24 Development activity

Here have been 3 major releases since Fido's first release in November 2010, which indicates a high level of activity.

5.25 Existing experience

Despite the interest in Fido by various memory institutions, experience with this tool is still limited (which is unsurprising for such a young tool).

5.26 Unidentified files

Although not part of the evaluation framework, it is useful to know something about which files can and cannot be identified by a tool. Fido was unable to identify 1385 files in the test dataset (some of these are objects that are nested inside a ZIP archive). The following table shows the file extensions that correspond to non-identified files:

Extension	Count	Remarks
fil	2	Text files with checksum values (Elsevier)
mol	3	MDL MOL file (chemical table format, http://en.wikipedia.org/wiki/MDL_Molfile)
oa3	69	XML metadata format
pl	3	Perl script
pm	4	Perl script
r	1	R script (R, programming language for statistical applications)
sgc	52	SGML metadata (Elsevier)
sgm	199	SGML metadata (Elsevier)
toc	59	Text table specific to Elsevier
xml	172	XML files without XML declaration
pdf	73	Portable Document Format
cif	10	Crystallographic Information Framework
txt	1	Text file
csv	3	Comma delimited text file
raw	737	Extension reserved for ‘raw bitmap’ format; in test dataset this extension is used for plain text files that are part of Elsevier metadata.

Note that these results cannot be directly compared to those of DROID, since DROID uses file extensions as a fallback mechanism if signature-based identification is not possible. Since Fido’s –*extension* switch didn’t turn out to work as it should, the above results are solely based on file signatures. The difference with respect to DROID is largely due to the contribution of the “.raw” files in the test dataset. These file *are* identified by DROID (based on their file extension), but wrongly (DROID assumes ‘raw bitmap’ format, whereas in reality they are plain text files). The 73 PDF files that could not be identified are more reason for concern, since PDF should *always* be identifiable using byte signatures. Fido may be using outdated file signatures (to this author’s knowledge the PRONOM signatures of a number of PDF versions have recently been updated by TNA), but since the signature file doesn’t contain any version information there’s no way to find out where the signatures are coming from (which illustrates the point made about this in the preceding section on event information).

5.27 Conclusions

The evaluation of Fido revealed a substantial number of problems. Some of these problems only occur in the Jython version, whereas others are specific to the Python version. The following issues apply to both these versions:

- Lack of any option to automatically convert information from PRONOM (or DROID signature files) to Fido format.
- Absence of any versioning information in Fido's signature and file extension files.
- Inability to identify Microsoft Office formats beyond the "OLE2 Compound Document Format" level.
- Limited provision of event information (similar to DROID 6).
- Reporting of MIME types doesn't appear to work.
- Reporting of *relative* file paths in the output file could be a problem in some cases (although this can be avoided by the user by always using full path references when invoking Fido).

Issues that are specific to the Python version of Fido (0.9.3) are:

- Undocumented behaviour of Fido's installation script, resulting in the installation of Fido in some Python system directories on Windows-based systems (this also applies to the Windows installer package).
- The batch files and shell scripts that are provided with Fido will –at best- only work when invoked from the directory in which these scripts are installed (and even then they will mostly fail because they contain references to nonexistent files and to the Python interpreter without specifying its full file path).
- Confusing and partially outdated information in the readme file about the use of the Fido command line.
- Ditto for Github site (which describes the command-line interface of a Fido version that has never been released as a Python version at all).
- Documentation describes method for invoking Fido that –in most cases- is unnecessarily verbose.
- The *–extension* switch does not work, and its activation results in a run-time error.

The following issues are specific to the Jython version (0.9.5):

- The software only works if it is called from the directory in which the JAR file is installed.
- Erroneous handling of file paths under Windows (not tested under Linux).
- Attempts to analyse one file object at a time all failed (empty output without any error messages).
- Recursing into (sub)directories doesn't work.
- Extremely poor computational performance: about 14 times slower than the Python version while analysing many files at a time.

Despite these problems, the Python version of Fido is a potentially interesting candidate for inclusion in the SCAPE architecture, provided that these issues (most of which are relatively small) get fixed. Apart from the code itself, the importance of accurate and up-to-date documentation, utility scripts that work and sensible behaviour of installation scripts should not be overlooked. In this regard the current situation makes things unnecessarily difficult and sometimes frustrating from a user's point of view, which will not help the acceptance and adoption of this tool. Fido's main strength is its performance while working at one file object at a time, which is about 35 times better than DROID 6. For many files at a time, the performance is similar to DROID. In addition, compared to DROID, Fido's command-line interface is somewhat better suited for use in automated workflows, since no additional data exporting actions are needed after the identification process.

For the Jython version things are slightly different. Apart from the fact that none of the tests with this version were particularly successful, its performance is also very poor, even much worse than DROID 6. This raises the question why anyone would need a Java-based version of Fido in the first place, especially with Python being available for nearly all popular platforms.

On a final note, it may be worthwhile to consider an upgrade of Fido to code that is compatible with the Python 3.x syntax. This is not an urgent issue (also, the Python 2.x range of interpreters is still widely used), but it might make things easier in the long run.

6 Unix File Utility

6.1 Overview

File is a command-line utility that is part of every major Unix and Unix-like operating system. The first version of this tool dates back to 1973. It identifies files based on signatures ('magic numbers') stored in a 'magic' file. An open source implementation of the tool exists, and it has been ported to other operating systems (e.g. Windows). The file utility reports identification results as MIME types. File performs three sets of tests: filesystem tests, magic tests, and language tests. The first test that succeeds terminates the utility.

The filesystem tests are based upon the results of a stat system call, and determine if the file is empty, or a special file, e.g. sym-link, or a named pipe.

The magic tests check for files with data in particular fixed formats. Any file with some invariant identifier at a small fixed offset into the file can usually be described in this way. The identifiers are read from a compiled magic file, present at one of a set of possible locations or passed as an argument to the utility.

If both the above tests fail the file is tested to see if it is a text file. ASCII, ISO-8859-x, non-ISO 8-bit extended-ASCII sets, UTF-8 encoded Unicode, UTF-16 encoded Unicode, and EBCDIC are tested for. If a text character set is identified a language check is performed using keywords, e.g. the presence of the word "struct" indicates a C program. Command line options can be used disable some of these tests.

All tests that are presented here were done with a Windows port of version 5.03 of the File utility which is part of the GnuWin package[16]. This version dates back to 2009. The most recent version of File is version 5.07, but for this version no ready-to-use Windows binaries are available (although it is possible to create these by compiling the source code).

6.2 Tool interface

File has a command line interface.

6.3 License type

File is released under the original 4-clause BSD License, which permits use, modification, inclusion in other products, and redistribution, but contains the controversial advertising clause.

6.4 Language

The file utility is written in C.

6.5 Platform dependencies

The file utility comes as standard with all major Unix and Unix-like (linux) operating systems. It is also available for Windows as a stand alone utility or as part of the Cygwin distribution, although it is not packaged as part of the basic Cygwin install. File is also included in OS-X according to its web site but we cannot confirm this.

[16] Link: http://gnuwin32.sourceforge.net/packages/file.htm

6.6 Coverage of file formats

A report by Underwood (2009) mentions that 'magic' file version 4.21 covers about 2000 file types, although that version dates back to 2008 or earlier (the current version is 5.07).

6.7 Extendibility

The file command uses the contents of magic files containing invariant identifiers and offsets to their position. New signatures can be added, compiled and tested by anybody. To get signatures added to the official magic files they need to be supplied in magic format to the maintainer, Christos Zoulas. However, the way in which the signatures are organised is rather clumsy: the 'magic' file is compiled from a set of over 200 separate smaller files which are stored in a directory called 'magdir' in the source distribution. The documentation of File doesn't describe how this is done, and apparently no explicit version management exists either. In fact, the 'Bugs' section of the File man page acknowledges that this is a problem by commenting that "[t]here must be a better way to automate the construction of the Magic file from all the glop in magdir"[17].

6.8 Output format

The file command outputs its results as plain text, usually in the format:

```
file_name ; identification text
```

The format of the identification text depends upon the arguments used to invoke file. Without arguments the output is free text and contains some basic characterisation information depending upon the file format (e.g. number of words). When invoked with the -i option the output is of the format:

```
file_name ; mime-type; character-encoding.
```

6.9 Unique output identifiers

Identification results are reported as MIME types (e.g. 'application/pdf') and character encoding. Alternatively, results may be reported as a textual description (e.g. 'PDF document, version 1.3').

6.10 Granularity of output

The use of MIME type as the primary identifier implies that there is no one-to-one match with the PRONOM or OPF registries. Since PRONOM does also contain MIME type information, a mapping to PRONOM is possible. However, PRONOM's PUID classification uses a level of granularity that is higher than the more general MIME type subdivision, which means that for many formats the mapping to PRONOM will result in a range of possible PUIDS for one single MIME type. An alternative would be to use File's textual descriptions, which provide a higher level of granularity.

6.11 Accuracy of reported results

Not analysed yet.

6.12 Comprehensiveness and completeness of reported results

Compared to DROID and Fido, the reporting of the analysis results is rather sparse (although it provides additional information on character encoding). As with Fido, File reports relative file paths

[17] Link: http://linux.die.net/man/1/file

if relative paths are used on the command line interface, but again this behaviour can be avoided by using full paths on the command line.

6.13 Fit to needs of preservation community

Unlike DROID and Fido, File is not specifically targeted at the preservation community. Also, its documentation states that “file uses several algorithms that favor speed over accuracy, thus it can be misled about the contents of text files”[18]. Inclusion of the tool in automated workflows would be quite straightforward.

6.14 Ability to deal with nested objects

File has an ‘--uncompress’ option, which (according to its documentation) causes the tool to try to look inside compressed files. A simple test on a ZIP file produced an abnormal (non-zero) return code with an accompanying “compressed file format not implemented” message. We didn’t do any further tests on other compressed formats, but overall it seems advisable *not* to use this switch in any workflows[19].

6.15 Ability to deal with composite objects

Formats such as Microsoft Word 97 and Open Document Format are based on multiple file objects that are held together by a physical container (e.g. OLE2 for Microsoft Word 97, and ZIP for Open Document Format). Some preliminary tests showed that Open Document Text files are identified as Open Document format (and not as plain ZIP, even though ZIP is the container format). Interestingly, an EPUB file (which also uses ZIP as a container) was identified as ‘application/octet-stream’, and not even as a regular ZIP file. MS Word files were identified as ‘application/msword’; however, one MS Excel spreadsheet was identified as ‘application/vnd.ms-office’ (whereas other Excel files in the test dataset were correctly identified as ‘application/vnd.ms-excel’). So overall the identification of these formats appears to be hit and miss. It should be noted here that a relatively old ‘magic’ file was used that dates back to early 2009, so things may have improved in newer versions. As we were not able to locate a pre-compiled version of a more recent ‘magic’ file we couldn’t follow up on this at this stage.

HTML represents another class of composite objects, where, for example, an individual HTML file refers to external style sheets and images, which are all needed for proper rendering. The main difference with e.g. Open Document Format or EPUB is the absence of a physical container file. File does not have any mechanism to recognise the interdependencies between the individual components of this particular class of ‘composite objects’.

6.16 User documentation

There are several different versions of the File command, and the website of its main developer doesn’t include any original documentation, but instead links to the manual page of the tool for the OpenBSD platform[20]. For several other platforms similar (but not quite identical) manual pages exist. Although this is somewhat confusing, the functionality and syntax for all versions of File that are part of any Unix distribution are quite strictly constrained by a technical standard[21] from the Open Group,

[18] See following link: http://www.openbsd.org/cgi-bin/man.cgi?query=file&apropos=0&sektion=1

[19] This applies in particular to many-files-at-a-time scenarios such as in Section 6.2.17, since File will exit upon the first encountered ZIP file in that case.

[20] Link: http://www.openbsd.org/cgi-bin/man.cgi?query=file&apropos=0&sektion=1

[21] See also: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/file.html

which is the certifying body for the UNIX trademark. As a result, they can all be used more or less interchangeably. In addition, File's --help switch lists all command line options.

6.17 Computational performance: one file at a time

6.17.1 KB Scientific Journals data set

The 'treeLaunch' tool was used to get an impression of File's performance when it is called to process one file at a time. It was set up to recursively traverse the KB Scientific Journals data set's directory tree, and run File for each encountered file object. For an individual file object this results in a command line like:

```
file --mime-type --mime-encoding --no-pad 1-1-70.pdf
```

It took slightly over 5 minutes to analyse the whole KB Scientific Journals data set in this way. Figure 6-1 and Table 6-1 summarise the main results. On average File needs less than 0.03 seconds per file object, with a maximum of about half a second. This means that for the one-file-at-a-time scenario File's overall performance beats DROID 6 by a factor of about 440. As with Fido, File shows a slight increase in processing time with file size (Figure 6-2).

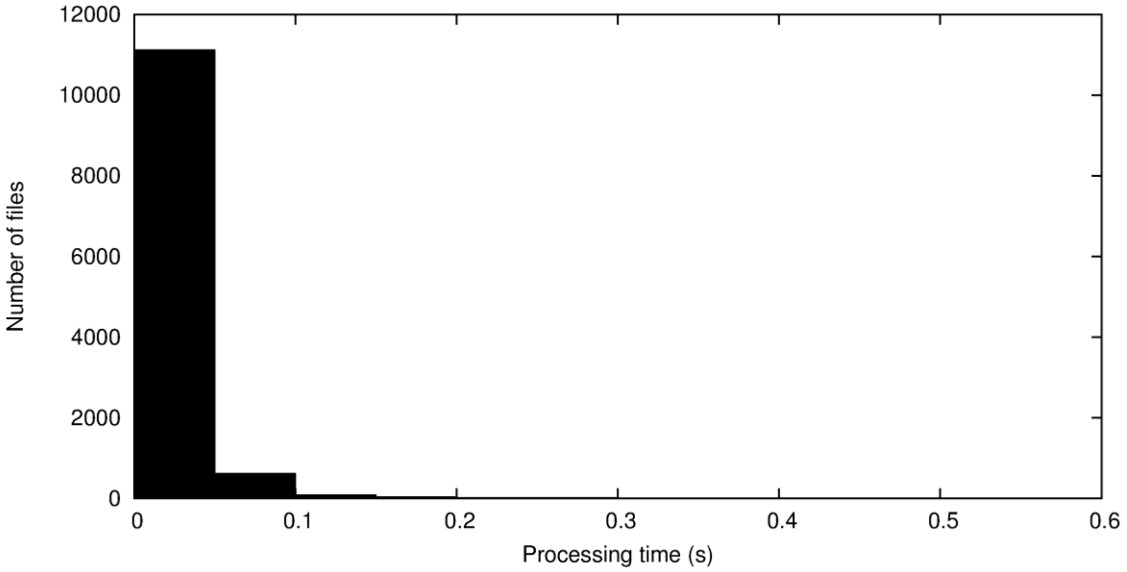

Figure 6-1 Distribution of processing time per file object for Unix File utility one-file-at-a-time scenario, KB Scientific Journals data set.

Table 6-1 Summary performance statistics for Unix File utility one-file-at-a-time scenario, KB Scientific Journals data set (expressed in seconds per file, except N). N=number of files; q1, median and q3 are 1st , 2nd and 3rd quantiles, respectively.

N	min	q1	median	mean	q3	max
11,892	0.01500	0.01600	0.01600	0.02706	0.03100	0.53100

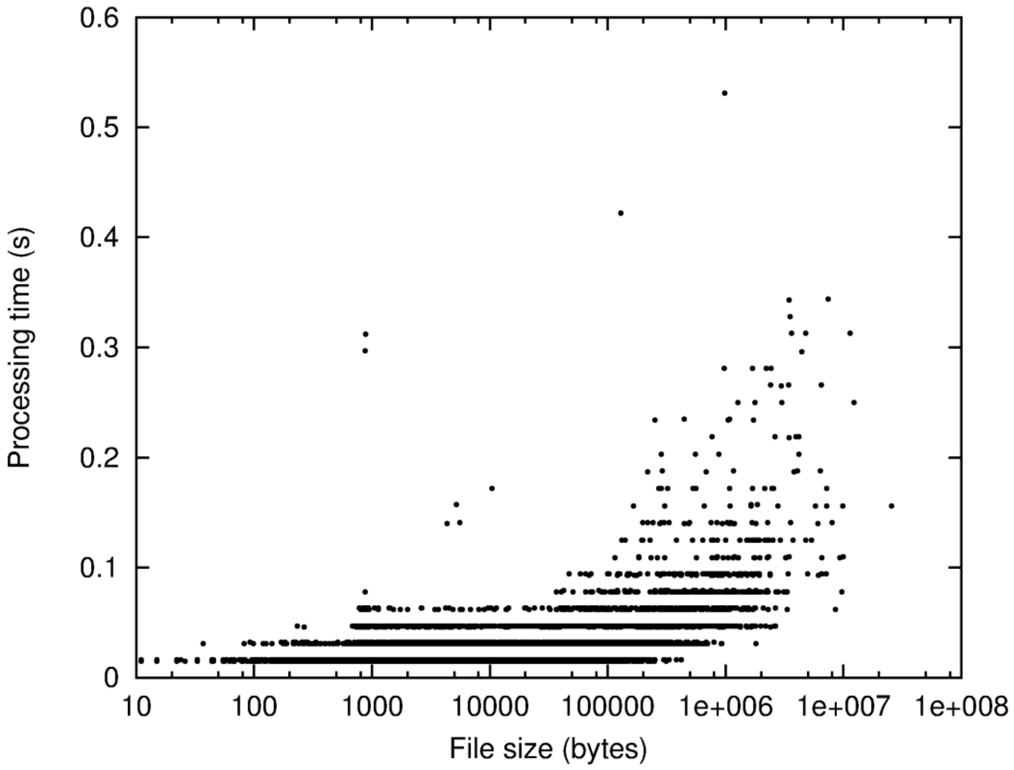

Figure 6-2 Scatter plot of processing time per file object versus file size for Unix File utility one-file-at-a-time scenario, KB Scientific Journals data set. Note logarithmic scale on horizontal axis.

6.17.2 KB Large data set

Table 6-2 shows the results of an additional test that was done on the KB Large data set (again using the 'treeLaunch' tool). The table demonstrates that in general file size does not affect processing time very much. Once again the TAR archive is an exception. However, unlike DROID or Fido, File does not analyse the contents of container files, which makes the increase in process time somewhat surprising in this case.

Table 6-2 Performance for individual files in KB Large data set

File name	Size (bytes)	Processing time (s)
dpo_tonal_00990.tiff	36,420,900	0.109
IMAGE000060_lossless_colour.jp2	49,818,414	0.031
IMAGE000060.TIF	114,664,502	0.047
SGD_19451955_0000002_ID371.pdf	325,101,508	0.047
DipAsset6984444754559678047.tar	534,968,320	0.422
KBDVD.iso	683,180,032	0.047
KBDVD17062011.img	693,993,472	0.031
DNEPABOstg.PST	1,895,515,136	0.156

6.18 Computational performance: many files at a time

To test File's performance while working on a large number of objects, we performed a recursive scan of all file objects in the directory structure of the KB Scientific Journals data set and measured the time needed for this. Unlike DROID or Fido, File doesn't have an in-built ability to recursively scan a directory structure. However, it is possible to specify all files that need to be analysed in a text file (using the '--files-from' switch). Therefore, we first generated this list using Windows' 'dir' command, and used this as input to File. This could also be done even more efficiently using pipes, which eliminates the need to create a temporary file. As the 'timeit' utility appeared to crash on most Windows system commands (including 'dir'), the temporary file approach was used instead here, using the following commands[22]:

```
dir /B /S /A-D D:\aipSamplesUnpacked > temp
timeit C:\Temp\identtools\file-5.03-bin\bin\file.exe --mime-type --mime-
encoding --no-pad --files-from temp > fileOut.txt
```

The first line calls the Windows 'dir' command with the options /B (bare format), /S (recursively walk through directory structure) and /A-D (don't display files that have the 'directory' attribute). Output is written to a temporary file. The second line calls File, using the temporary file as input. Unlike the DROID and Fido experiments, this will not recurse inside ZIP archives (see above for the reason for this). Also note that the 'dir' command was excluded from the performance measurement due to an incompatibility issue with the timer tool. However, the total added overhead of this first step is negligible (about 2 seconds).

The result for the second step (File) is:

Elapsed Time:	0:01:52.311

This means that File needs about 2 minutes to scan 1.15 GB of data in the used test environment. This is better than both DROID 6 and Fido (although the results are not 100% comparable since those tools also looked into ZIP archives).

6.19 Stability

Not yet tested explicity for SCAPE but File is used:

- In nearly all BSD distributions
- In nearly all Linux distributions
- libmagic (File's underlying library) is used by apache httpd servers mod_mime_magic module

This represents a significant user community. Reported bugs are generally fixed quickly. Also, the current tests did not result in any crashes or other stability issues.

6.20 Error handling and reporting

The file utility returns 0 upon success and non zero on failure.

[22] The equivalent command using pipes would be:

```
dir /B /S /A-D D:\aipSamplesUnpacked | C:\Temp\identtools\file-5.03-bin\bin\file --
mime-type --mime-encoding --no-pad --files-from - > fileOut.txt
```

6.21 Provision of event information

The version of the file command can be obtained by using the -v argument, which also returns the location of the magic file (but not its version, which is unsurprising given that no versioning system appears to exist).

For file objects that cannot be identified File returns MIME type 'application/octet-stream'. Zero-byte files result in MIME type 'application/x-empty'. If File is instructed to analyse a file object that doesn't exist, it will report this as follows:

```
nonexistentFile.dat; cannot open `nonexistentFile.dat' (No such file or
directory)
```

6.22 Maturity and development stage

The original version of file originated in Unix Research Version 4 in 1973. All major BSD and Linux distributions use a free, open-source reimplementation that was written in 1986-87 by Ian Darwin from scratch. It was expanded by Geoff Collyer in 1989 and since then has had input from many others. From late 1993 onwards its maintenance has been organized by Christos Zoulas.

6.23 Development activity

During the first 6 months of 2011, thee major versions (5.05, 5.06 and 5.07) of File have been released[23]. The tool's long history and its inclusion in so many operating systems make it extremely unlikely that development will cease in the foreseeable future.

6.24 Existing experience

Since File has been around for such a long time and its use is so widespread, this means that there is a large user community that has extensive experience with this tool. Less is known about its use in the archival community, although File is included in the FITS tool set (which is evaluated in Chapter 7 of this report).

6.25 Unidentified files

Although not part of the evaluation framework, it is useful to know something about which files can and cannot be identified by a tool. File was unable to identify 22 files in the test dataset. They were all identified as 'application/octet-stream'. These were mainly plain text files with a .raw extension which are part of Elsevier's metadata. In addition, one TIFF image could not be identified. However, upon closer inspection it turned out that this file was probably corrupted[24]. A cursory scan of File's output also revealed that some plain text files were mistakenly identified as C++, Fortran, Lisp and Pascal source code.

6.26 Conclusions

The evaluation of the File utility revealed a number of particular strengths as well as some potential problems.

Its main strengths are:

[23] See following link: ftp://ftp.astron.com/pub/file/

[24] Interestingly, this file was judged to be 'well-formed and valid' by JHOVE 1.4; however, upon opening in IrfanView the image turned out to be mostly blank, and a further inspection in a Hex editor revealed suspicious amounts of null-bytes.

- Excellent computational performance, both for one- and many- file-at-a time use cases
- Maturity, stability and width of user base
- Large number of supported formats (although any documentation on *which* formats are supported doesn't exist, and this information cannot be easily extracted from the 'magic' file either)

Potential problems for use in preservation workflows are:

- Management of file signatures (especially creation and updating of the 'magic' file from hundreds of smaller files, of which we were not able to track down any documentation)
- Lack of any versioning information in the 'magic' file
- One-to-one mapping of MIME types to PRONOM identifiers not possible for every format
- Mapping to textual descriptions would overcome this to a large extent, but creating such a mapping may be quite cumbersome.
- Identification of text-based formats can be unreliable (although based on the current analyses it is not possible to say if DROID or Fido are any better!)
- No scanning inside ZIP archives
- Latest binaries are not always available for all major platforms (e.g. Windows), so it may be necessary to build these from the source code

Despite these problems, File may nevertheless be of considerable value to the SCAPE architecture. Compared to DROID and Fido, it results in a considerably lower number of unidentified files (although the results are not completely comparable because of the non-inclusion of file objects that are embedded in ZIP archives). One possibility would be to use File as a primary identification tool, and use DROID and/or Fido in addition for any formats that are not handled by File in sufficient detail.

7 FITS (File Information Toolset) 0.5

7.1 Overview

FITS is an acronym of ‘File Identification Tool Set’. It is not a stand-alone characterisation tool, but rather a wrapper around a number of external tools. Currently, these are:

- Jhove (AKA Jhove1)
- Exiftool
- National Library of New Zealand Metadata Extractor
- DROID 3.0
- FFIdent
- Unix File Utility (windows port)

In addition, FITS supplies two original tools: FileInfo and XmlMetadata. It is also possible to add other tools. FITS converts all native output from these underlying tools into a common format (FITS XML). In addition, FITS is capable of handling conflicting output of these tools in a number of ways. FITS is developed by the Harvard University Library Office for Information Systems, and it is released as open source software. Evaluated here is version 0.5, which was released in February 2011. Apart from identification, FITS can also be used for feature extraction and validation. This report only addresses the identification functionality.

7.2 Tool interface

FITS has a command line interface, which can accessed using a batch file (Windows) or a shell script (Linux/OS X). The batch file only works if it is invoked directly from the application directory (i.e. the directory in which the batch file is installed). Unlike DROID (which has a similar issue), in the case of FITS there doesn’t appear to be any easy way of getting around this. Alternatively, the FITS functionality may also be accessed through a Java application programming interface (API).

7.3 License type

FITS is released under a GNU Lesser GPL license. However, the tools that are bundled with FITS are released under a number of different license types. The same applies to a number of libraries that are used by FITS.

7.4 Language

FITS is written in Java.

7.5 Platform dependencies

According to the user documentation FITS runs on Windows, Linux and OS X (Mac). Furthermore Java 1.6 or higher is required.

7.6 Coverage of file formats

Depends on wrapped tools. Both DROID and the Unix File Utility are wrapped by default, so the format coverage is at least the union of the formats that are recognised by these tools .

7.7 Extendibility

In principle any type of tool can be added to FITS. The FITS output format also allows the use of 'external identifiers', which means that any kind of format identifier can be used.

7.8 Output format

FITS reports its output to FITS XML format[25]. In addition, an option exists to convert FITS output to a standard metadata schema (e.g. textMD for text files or NISO/MIX for image files). As this option is mainly relevant for feature extraction, it will not be further discussed here.

7.9 Unique output identifiers

By default, FITS reports the identification results in terms of a (textual) format description (e.g. "Plain text") and MIME type. Depending on the tool that was used for the identification the identification results may also contain one or more so-called "external identifiers", which are the native identifiers of the wrapped tools. For instance, if one of the tools that were used to identify a particular file was DROID, the FITS output file will contain one or more external identifier elements that contain the corresponding PUIDs. Note however that the FITS output will not contain any PUID information at all for any files that are not recognised by DROID.

7.10 Granularity of output

Mapping to PRONOM and the OPF registry is possible if FITS output file contains PUID information. However, PUIDs are not used as the primary identifier, and the FITS output may not contain any PUIDs at all.

7.11 Accuracy of reported results

Not analysed yet.

7.12 Comprehensiveness and completeness of reported results

The information in the FITS output file is very comprehensive. It contains the status of the identification ("SINGLE_RESULT", "CONFLICT" or "UNKNOWN)", the format name (as a textual description), MIME type, and the name and version of the tool that was used to establish this information. Also, FITS provides comprehensive event information (see also Section 7.21).

7.13 Fit to needs of preservation community

FITS is specifically targeted at the preservation community (and it is also being developed by a member of that community). The comprehensiveness of its output, and the emphasis on providing event information (see Section 7.21) make FITS particularly suited for deployment in automated workflows.

7.14 Ability to deal with nested objects

FITS 0.5 is not able to look inside ZIP files (or other file archive formats).

7.15 Ability to deal with composite objects

Some limited test on a number of composite formats revealed the following:

- Microsoft Word and Excel files are identified correctly, although without any indication of the specific format version.

[25] The schema of FITS XML is located here: http://hul.harvard.edu/ois/xml/xsd/fits/fits_output.xsd

- Any attempts at analysing OpenDocument Text files resulted in an exception being thrown from NLNZ Metadata Extractor (tried this on several). Output file was written nevertheless. The ODT files were also identified correctly.
- An EPUB file was erroneously identified as an OpenDocument Text file. This is not completely surprising, as EPUB uses the Open Container Format (OCF), which describes the packaging of individual file components of an EPUB publication in a ZIP archive. OCF is in turn based on the Open Document Format (of which ODT is a sub-format). Incidentally FITS did not crash on this particular file. Another EPUB file was identified as an ordinary ZIP file.

HTML represents another class of composite objects, where, for example, an individual HTML file refers to external style sheets and images, which are all needed for proper rendering. The main difference with e.g. Open Document Format or EPUB is the absence of a physical container file. FITS does not currently have any mechanism to recognise the interdependencies between the individual components of this particular class of 'composite objects'.

7.16 User documentation

Documentation is provided as an online 'FITS User Guide'[26]. The information in the User Guide is adequate for getting started with FITS, although for serious (e.g. operational) uses of the tool it is somewhat lacking in detail. This applies in particular to the explanation of the output format.

As an example, for file objects that cannot be identified, FITS reports the identification status attribute as "UNKNOWN". According to the User Guide, this attribute can only have a value of "SINGLE_RESULT" or "CONFLICT" (without any mention of "UNKNOWN")! In fact, the "UNKNOWN" value is not included in the FITS output schema either, which means that the output file is not valid according to its own schema [27] (this was confirmed by a check using an XML validator tool).[28]

Also, as described in Section 7.9, by default FITS reports the identification results in terms of a (textual) format description (e.g. "Plain text") and MIME type. However, from the documentation it is not clear where either the format descriptions or the MIME type strings are originating from. An inspection of the FITS directory structure reveals that this is done using a number of style sheets (one for each tool). Although this information is not necessarily needed to run FITS, having some general idea of where the descriptions and mime types are coming from is certainly helpful in interpreting the results[29].

A final example: FITS uses the concept of a 'format tree' to handle formats that are a subset of a more general format. Although the general idea is mentioned in the User Guide, a more detailed description of the specific format of the format tree is currently missing. Moreover, the format tree is defined in terms of textual format descriptions that are not documented anywhere (see above). Users of the software who wish to wrap their own tools in FITS will however need this information.

[26] Link: http://code.google.com/p/fits/wiki/user_guide

[27] A commenter to the on-line User Guide reports a similar issue where the status attribute is "PARTIAL". See following link (bottom of page): http://code.google.com/p/fits/wiki/general

[28] After reporting the above issue on the FITS project website we got an almost immediate response from one of the developers, who instantly produced a corrected version of the XML schema. So it would be safe to assume that the schema-related issues will be fixed in an upcoming release.

[29] Just an example: since this type of output is so similar to the output of the Unix File tool, this could easily lead to the erroneous assumption that FITS simply copies the identification results of 'File' (which is not the case!). So explaining this in the documentation would be helpful to avoid such misunderstandings.

7.17 Computational performance: one file at a time

7.17.1 KB Scientific Journals data set

The ‘treeLaunch’ tool was used to get an impression of the performance of FITS when it is called to process one file at a time. It was set up to recursively traverse the KB Scientific Journals data set’s directory tree, and run FITS for each encountered file object. For an individual file object this results in a command line like:

```
fits -i 1-1-70.pdf
```

It took slightly less than 21 hours to analyse the whole KB Scientific Journals data set in this way. Figure 7-1 and Table 7-1 summarise the main results. On average FITS2 needs 6.3 seconds per file object, with a maximum of 23.7 seconds. For the one-file-at-a-time scenario this is almost twice as fast as DROID 6, but compared to Fido or the Unix File utility this is still quite slow. Processing time appears to increase quite markedly with file size (Figure 7-2). One noteworthy feature of Figure 7-2 is that it shows some marked clustering around a number of narrowly-defined time intervals. These clusters can also be seen as minor ‘peaks’ in Figure 7-1, and appear to be caused by the invocation of format-specific tools (mostly of the files in these clusters are XML).

The ‘issues’ list at the FITS project Wiki also contains an entry about the tool’s processing performance[30]. In a response to this, Spencer McEwen comments that in his experience JHOVE and NLNZ Metadata Extractor take the longest amount of time, and read much more data from the analysed files than the other tools. So, we repeated the performance test with these two tools disabled. Performance was somewhat better in this case, but FITS still needed about 17 hours to process the dataset.

[30] Link: http://code.google.com/p/fits/issues/detail?id=20

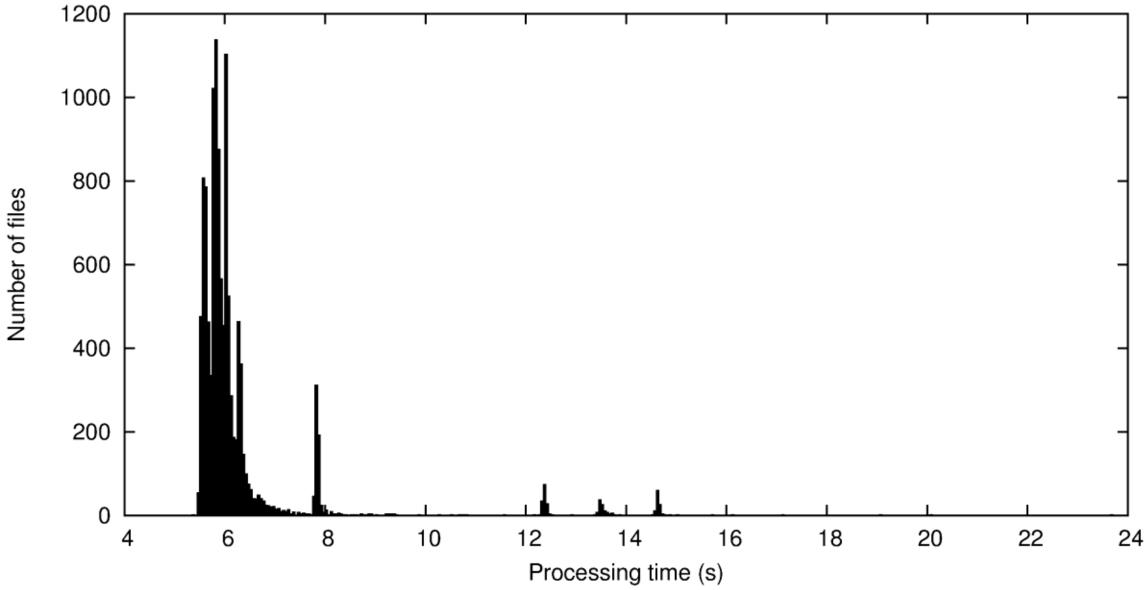

Figure 7-1 Distribution of processing time per file object for FITS one-file-at-a-time scenario, KB Scientific Journals data set.

Table 7-1 Summary performance statistics for FITS one-file-at-a-time scenario, KB Scientific Journals data set (expressed in seconds per file, except N). N=number of files; q1, median and q3 are 1st , 2nd and 3rd quantiles, respectively.

N	**min**	**q1**	**median**	**mean**	**q3**	**max**
11,892	5.390	5.750	5.891	6.269	6.156	23.688

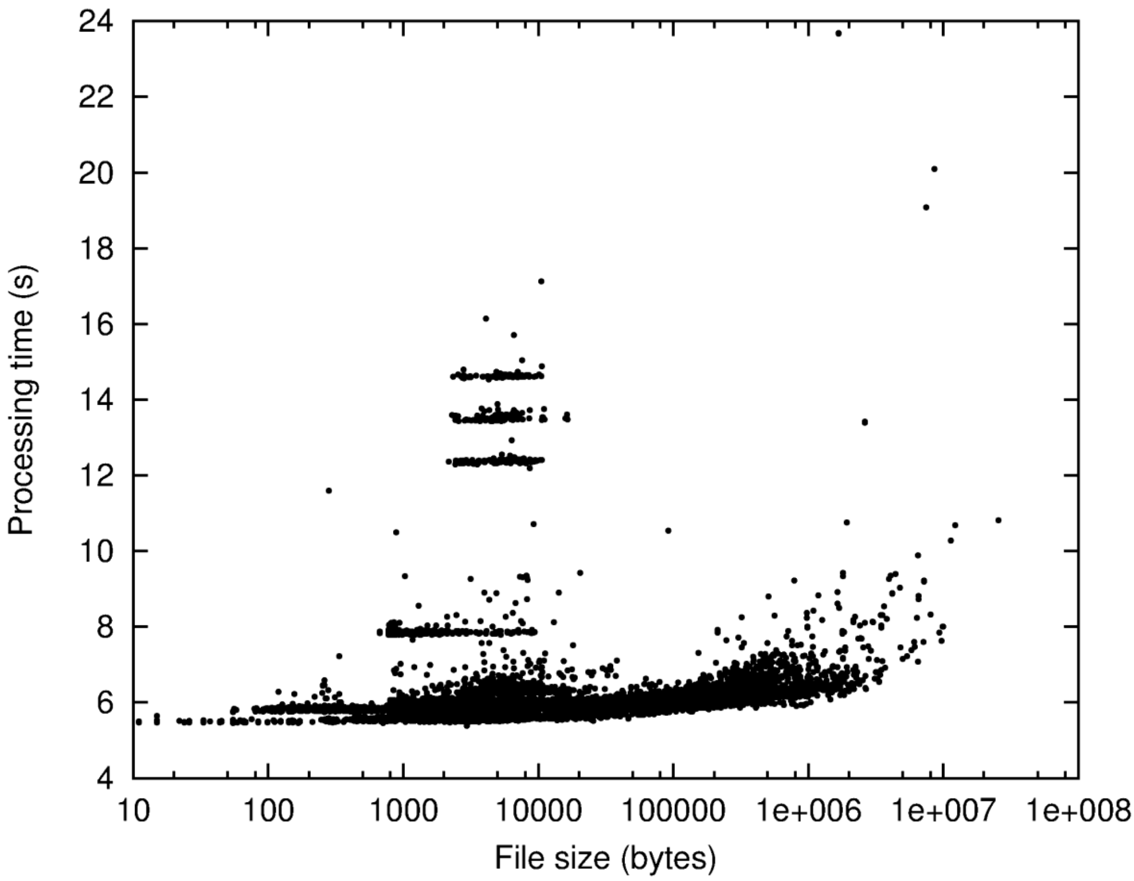

Figure 7-2 Scatter plot of processing time per file object versus file size for FITS one-file-at-a-time scenario, KB Scientific Journals data set. Note logarithmic scale on horizontal axis.

7.17.2 KB Large data set

Table 7-2 shows the results of an additional test that was done on the KB Large data set (again using the 'treeLaunch' tool). The table shows that processing time is strongly related to file size. Figure 7-3 illustrates this graphically.

Table 7-2 Performance for individual files in KB Large data set

File name	Size (bytes)	Processing time (s)
dpo_tonal_00990.tiff	36,420,900	12.828
IMAGE000060_lossless_colour.jp2	49,818,414	8.563
IMAGE000060.TIF	114,664,502	11.172
SGD_19451955_0000002_ID371.pdf	325,101,508	156.003
DipAsset6984444754559678047.tar	534,968,320	64.079
KBDVD.iso	683,180,032	72.361
KBDVD17062011.img	693,993,472	78.048
DNEPABOstg.PST	1,895,515,136	150.972

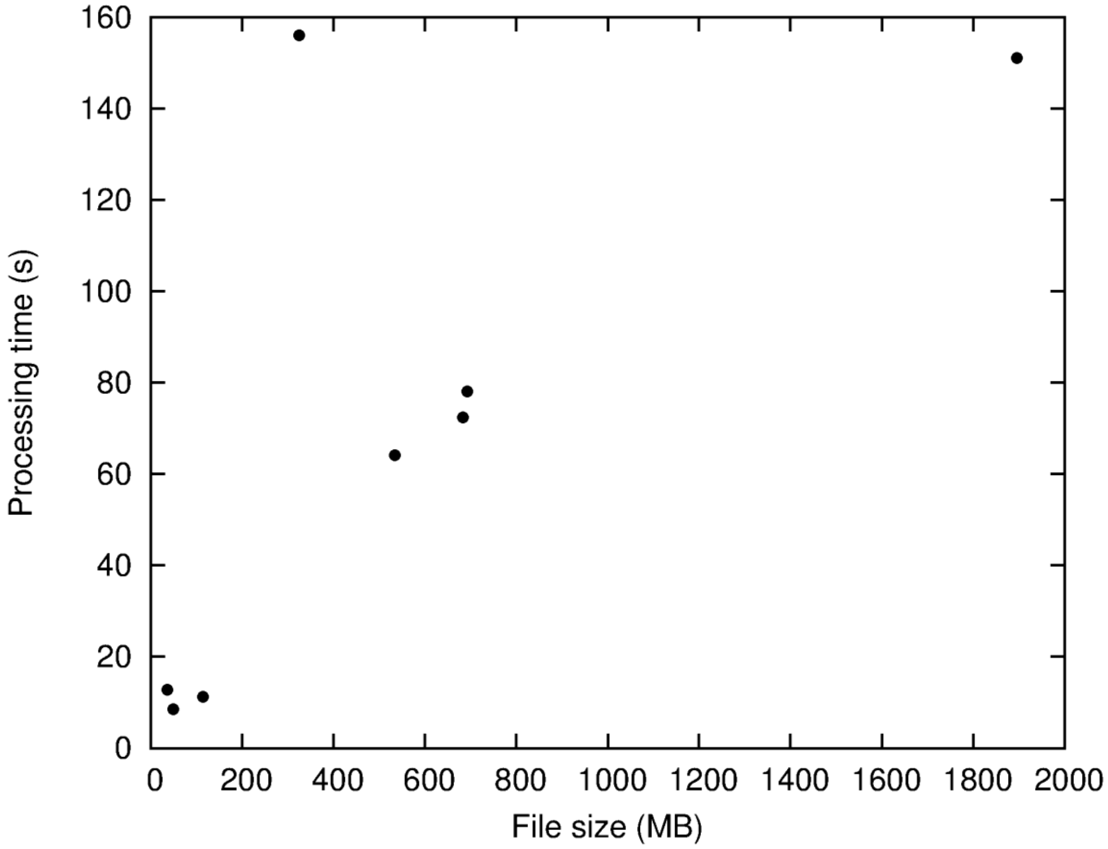

Figure 7-3 Scatter plot of processing time per file object versus file size for FITS one-file-at-a-time scenario, KB Large data set.

7.17.3 Additional tests on influence of FITS configuration

FITS is different from most of the other tools covered by this report in that it wraps around multiple tools, which all add up processing time. It is possible to switch individual tools on or off through FITS' configuration files. In order to get an indication of the impact of this on FITS' performance, we performed an additional test on a small set of 5 files. Performance was measured for two different FITS configurations:

- FITS' standard configuration.
- A 'minimal' configuration, in which all tools except the Unix File tool are disabled.

Table 7-3 shows the results.

Table 7-3 Processing time for some format-size combinations using standard and 'minimal' FITS configuration.

Format	Size	Processing time (s), standard configuration	Processing time (s), 'minimal' configuration
PDF	25 MB	10.5	2.2
PDF	69 KB	6.2	2.3
JPEG	69 KB	6.1	2.3
ZIP	8 MB	20.3	2.4
MS Word	2.5 MB	13.1	2.3

The table shows that the processing time per file decreases quite dramatically using the 'minimal' FITS configuration. Despite this, the overall performance doesn't even come close to using the Unix File tool in stand-alone mode, and FITS is adding quite a bit of overhead of its own (which is probably due to initialisation).

7.18 Computational performance: many files at a time

Unlike DROID and Fido, FITS 0.5 does not have any functionality for processing multiple files in one run.

7.19 Stability

Not investigated in detail so far. FITS did raise an exception while attempting to analyse a non-existent file. In addition various exceptions were raised by the NLNZ Metadata Extractor tool, but these didn't lead to any subsequent FITS exceptions.

7.20 Error handling and reporting

The FITS User Guide is not specific on this. It seems that any error messages are written to the console's standard error device.

7.21 Provision of event information

The FITS output file contains extensive event information, including:

- FITS version number
- A time stamp
- Names and version numbers of all tools that were used in the identification process
- Status fields that indicate whether output of different tools give conflicting information
- Status field that indicates whether a file could be identified at all
- Provenance of identification results (i.e. specific tool and tool version of each result)

One omission is that FITS does not report any information on the version of the used version of the DROID signature file (which is a particularly important piece of information). This issue aside, FITS' provision of event information is superior to any of the other tools that have been investigated in this report.

7.22 Maturity and development stage

The first version of FITS (0.2.5) was released in (late?) 2009. So far there have been nine releases, four of which were major releases. Although the documentation makes no mention of the current development stage of FITS, the number of the most recent release (0.5) suggests that the software is currently in beta stage.

7.23 Development activity

With seven releases between January 2010 and February 2011, development appears to be quite active.

7.24 Existing experience

Although FITS is fairly well-known within the preservation community, the degree to which it is used by memory institutions is largely unknown.

7.25 Unidentified files

No files were left unidentified. A number of text files with ".raw" extension were wrongly identified as "raw bitmap" (source: DROID).

7.26 Conclusions

The evaluation of FITS 0.5 reveals a mixed picture. Its main strengths are:

- Its ability to combine the identification functionality of several tools, and normalise the output of these tools.
- The possibility to add new tools
- It supports a large number of file formats.
- Comprehensiveness of the output files.
- Use of XML (and an associated schema) for output enables validation of the output files
- Provision of detailed event information (except the DROID signature file version, which is strangely absent). This makes FITS closely tailored to the specific needs of digital repositories.
- The behaviour of FITS is highly configurable (e.g. tools can be disabled, even for specific file extensions).
- The FITS 'format tree' provides a sensible mechanism for ensuring that, in case of multiple identification results by different tools, the most specific result is used.

In addition, FITS also offers the option to normalise its output according to standard metadata schemas such as NISO/MIX or textMD (although this is largely outside of the scope of this document).

However, there are some problems too:

- Most importantly, FITS is slow (although calling FITS from the Java API will most likely result in better performance than what was reported here).
- There is a lack of any option to process many files in one FITS run (which would improve performance by reducing initialisation times).
- The batch launcher only works when it is executed from the FITS installation directory, which is both clumsy and unnecessary.
- The User Guide provides enough information to get started with FITS, but for any "serious" (e.g. operational) applications it is lacking in detail.
- This also applies to the documentation of the configuration options: even though almost everything in FITS is highly configurable, the current documentation is not sufficient to fully take advantage of this.
- The FITS identification results are reported as textual format descriptions, which are currently not documented anywhere.

- One-to-one mapping of results to PRONOM identifiers is not possible for every format[31].
- No scanning inside ZIP archives.

Of the above problems, FITS' performance is the most serious one. However, it is important to stress that the results of FITS cannot be directly be compared to those of the other tools in this report. The main reasons for this are:

1. Unlike those other tools, each FITS run involves the invocation of multiple underlying tools, and each tool adds to the total processing time.
2. Identification is only a small part of FITS' functionality: it also includes feature extraction and validation. Computationally such operations are more 'costly' than simple identification, because files may need to be read/analysed in their entirety.
3. FITS has no built-in functionality for analysing many file files at a time, which means that the results for this use case had to be obtained by repeating the one file at a time use case for all file objects in the dataset.
4. The actual performance of FITS is affected by its configuration: performance can be improved by disabling specific tools.

As for the last point: even with JHOVE and NLNZ Metadata Extractor (which are supposed the slowest tools that are wrapped inside FITS) disabled, performance was still very poor in the tests. This casts some doubts on the suitability of FITS for processing large volumes of data in an operational setting.

On a final note, performance is likely to be better if FITS is invoked through its Java API (instead of the command line). Additional tests are necessary to confirm this.

[31] This is not really a limitation of FITS: some formats simply do not have an associated PRONOM identifier, and for tools that do not use some other identifier there is no mapping back to PUID.

8 JHOVE2

8.1 Overview

JHOVE2 is the successor of the well-known JHOVE tool. Its creators refer to JHOVE2 as a "Java framework for next-generation format-aware characterization". It is being developed by the California Digital Library, Portico, and Stanford University, with funding from the Library of Congress under its National Digital Information Infrastructure and Preservation Program (NDIIPP). JHOVE2's functionality comprises identification, feature extraction, validation and policy-based assessment. The current analysis is restricted to JHOVE2's identification functionality. JHOVE2 is developed as open-source software. We evaluated JHOVE 2.0.0 (released April 2011).

8.2 Tool interface

JHOVE2 has a command line interface, for which a Windows batch file and a Unix shell script are provided. The batch file ("jhove2.cmd") contains the following reference to a configuration script that sets up a number of environment variables ("env.cmd"):

```
call env
```

This only works if the batch file is launched from its installation direction, and will go wrong otherwise (unless the installation directory is included in Windows' 'path' environment variable).

This can be fixed by changing the call to:

```
call %~dp0\env
```

This inserts the path to the directory where "jhove2.cmd" is installed[32].

8.3 License type

JHOVE2 is released under an open-source BSD License. JHOVE2 uses a number of third party components, which are all released under a variety of open source license types.

8.4 Language

JHOVE2 is written in Java (version 6).

8.5 Platform dependencies

According to the JHOVE2 User Guide the software is designed to work with any implementation that is fully compliant with Java 6. The developers also state that the software has been tested on Sun's implementation of Java 6 on Windows, Solaris and Linux, and on Apple's implementation of Java 6 on Mac OS [JHOVE2a].

8.6 Coverage of file formats

JHOVE2 uses DROID 4.0 for identification[33]. DROID's identification is based on file signatures, or, alternatively, known extensions. These are defined in a signature file, which is regularly updated by

[32] This issue will be corrected in the next code release (Stephen Abrams, pers. comm.)

[33] Strangely JHOVE2's documentation doesn't appear to mention the included version of DROID, and the DROID license file that is included with JHOVE2 is for DROID 3.0.

The National Archives. Strangely the JHOVE2 install contains a very old version (20) of the DROID signature file (the current version is 49!). The DROID version that is used by JHOVE2 doesn’t support ‘container signatures’ for container formats such as ZIP (Open Document and Microsoft Office Open XML formats) and OLE2 (container for the Microsoft Office formats).

8.7 Extendibility

Since the number of formats that DROID can handle is defined by the information in the signature file, new formats can be added by modifying the signature file (or downloading the latest version from the National Archives). However, the configuration file of JHOVE2’s identification module contains a comment stating that “[t]he DROID Signature and Configuration files in the JHOVE2 distribution have been edited to, among other things, detect additional signatures for formats”. The nature and extent of this “editing” is not further specified, which raises some concerns on what happens if a user replaces the default signature file (which is very old) by a more recent one.

Another relevant feature of JHOVE2 is that external tools can be wrapped. In principle this would make it possible to incorporate additional identification tools into the JHOVE2 framework. Apart from this, individual modules of JHOVE2 can also be switched off (e.g. the policy-based assessment module, or the module that calculates message digests).

8.8 Output format

JHOVE2 can report output in text, JSON and XML format. The output format can be set with a command-line switch. Output is sent to the standard output device (default) or to a user-defined file. In all cases the output is extremely verbose (including detailed information on JHOVE2 sub-processes). As JHOVE2’s current documentation doesn’t cover its output, making sense of the output files can be quite a challenge. JHOVE2’s functional requirements suggest that the current output format is only intended as an intermediate format that can be converted to any desirable form using an XSL style sheet transformation[34]. The JHOVE2 Wiki contains a note that style sheets to transform JHOVE2 output to METS and PREMIS format will be included with release 2.1.0 of the software[35]. On a side note, it *is* possible to configure JHOVE2’s output. The procedure is explained in the section “Configure Unit and Displayer Properties” of the User’s Guide, but it involves editing numerous Java properties settings files. This is something that may be too complicated for most general users (besides, the description of the configuration directory structure doesn’t match the actual directory structure, which adds to the confusion).

8.9 Unique output identifiers

Neither the User’s Guide nor any of the separate module documentation documents give any information on how the identification results are reported. An inspection of some JHOVE2 output files[36] first reveals some identification-related information at the start of the file:

```
PresumptiveFormats:
  PresumptiveFormat {FormatIdentification}:
   NativeIdentifier {I8R}:
    Namespace: PUID
    Value: fmt/95
```

[34] Link: https://bytebucket.org/jhove2/main/wiki/documents/JHOVE2-functional-requirements-v1_4.pdf , Imperative Requirement 5 (page 6)

[35] See also: https://bitbucket.org/jhove2/main/wiki/StyleSheets

[36] For the sake of readability all output examples here are given in plain text format.

```
   JHOVE2Identifier {I8R}:
    Namespace: JHOVE2
    Value: http://jhove2.org/terms/format/pdf
   IdentificationProduct {I8R}:
    Namespace: JHOVE2
    Value: http://jhove2.org/terms/reportable/org/jhove2/module/identify/DROIDIdentifier
   Confidence: Tentative
```

Here we can see that JHOVE2 uses two identifiers:

- A PUID (fmt/95, which is PDF/A-1a)
- A "JHOVE2Identifier". In the above example its value is:
  http://jhove2.org/terms/format/pdf
  Although the JHOVE2 identifier has the format of a URL, it doesn't point to any existing document(s) on the JHOVE2 website.

We were initially puzzled by what seemed like a second block of identification-related information (including a list of PUIDs) further down the output file under the *Module {BaseFormatModule}* heading:

```
  Module {BaseFormatModule}:
   ModuleNotFoundMessage: [ERROR/PROCESS] No module name found for Identifier
http://jhove2.org/terms/format/pdf
   Format:
    Name: PostScript
    Identifier {I8R}:
     Namespace: JHOVE2
     Value: http://jhove2.org/terms/format/pdf
    AliasIdentifiers:
     AliasIdentifier {I8R}:
      Namespace: JHOVE2
      Value: http://jhove2.org/terms/format/pdf
     AliasIdentifier {I8R}:
      Namespace: PUID
      Value: fmt/14
     AliasIdentifier {I8R}:
      Namespace: PUID
      Value: fmt/15
     AliasIdentifier {I8R}:
      Namespace: PUID
      Value: fmt/16
     AliasIdentifier {I8R}:
      Namespace: PUID
      Value: fmt/17
     AliasIdentifier {I8R}:
      Namespace: PUID
      Value: fmt/18
     AliasIdentifier {I8R}:
      Namespace: PUID
      Value: fmt/19
     AliasIdentifier {I8R}:
      Namespace: PUID
      Value: fmt/20
     AliasIdentifier {I8R}:
      Namespace: PUID
      Value: fmt/95
```

The meaning of this information is not completely clear. Since fmt/14 to fmt/20 represent PDF versions 1.0 to 1.6 this might be some meta-information that is related to JHOVE2's "BaseFormat" module (e.g. al PUIDs that correspond to JHOVE2 identifier "http://jhove2.org/terms/format/pdf"). However, since the User's Guide doesn't contain any explanation of JHOVE2's output, users are left in the dark on this, which is not particularly helpful and rather confusing[37]. Also note that in the above example there is a mention of the PostScript format[38].

8.10 Granularity of output

The use of PUIDs ensures that JHOVE2's output can be mapped directly to the PRONOM registry (as well as the OPF registry).

8.11 Accuracy of reported results

Not analysed yet.

8.12 Comprehensiveness and completeness of reported results

The reporting of the analysis results is extremely comprehensive. There are two problems here:

1. JHOVE2's output format is undocumented. This applies both to the information units as well as the overall structure.
2. JHOVE2's output is extremely verbose (including detailed event information at the level of individual modules)

Because of the combination of 1 and 2 above it is quite difficult to make sense of the output in its current form (which was also illustrated in the previous section). However, JHOVE2's functional requirements suggest that the current output format is only intended as an intermediate format that can be converted to any desirable form using an XSL style sheet transformation[39]. The JHOVE2 Wiki contains a note that style sheets to transform JHOVE2 output to METS and PREMIS format will be included with release 2.1.0 of the software[40].

8.13 Fit to needs of preservation community

The preservation community is JHOVE2's primary target audience. Also, the JHOVE2 Project Team partners as well as JHOVE2's main funder (Library of Congress) are all actively involved in digital preservation. JHOVE2's functional requirements explicitly state that "[t]o the fullest extent possible, JHOVE2 should be easily integrated into existing workflows."[JHOVE2b].

8.14 Ability to deal with nested objects

JHOVE2 is specifically designed to deal with nested objects through its concept of *source units*, which are "entities that can be independently characterized" [JHOVE2a]. Nested objects are simply one class of aggregate source units. JHOVE2 currently recognises the following cases:

- A file system directory
- A ZIP or TAR file
- Directories within an archive file.

[37] Update from Stephen Abrams in response to this issue: 'The first block (under "PresumptiveFormat") documents the specific format (fmt/95, PDF/A) returned by the identification tool (i.e. DROID). The second block (under "Module") documents the full set of information JHOVE2 has been configured with regarding this format. Since we do not yet have a PDF module, we consider all of the PDF version PUIDs to be alias of PDF/A. When the PDF module becomes available later this year, we will be more precise in distinguishing the versions and associated more tightly bound aliases.'

[38] This turns out to be a bug (Stephen Abrams, pers. comm.)

[39] Link: https://bytebucket.org/jhove2/main/wiki/documents/JHOVE2-functional-requirements-v1_4.pdf , Imperative Requirement 5 (page 6)

[40] See also: https://bitbucket.org/jhove2/main/wiki/StyleSheets

- A wrapper file (e.g. a TIFF file can be a wrapper for ICC profiles or XMP metadata)

JHOVE2 can deal with any combination of the above, provided that a module exists for the container/wrapper format.

8.15 Ability to deal with composite objects

Formats such as Microsoft Word 97 and Open Document Format are based on multiple file objects that are held together by a physical container (e.g. OLE2 for Microsoft Word 97, and ZIP for Open Document Format). In principle, these could all be treated as aggregate source units in JHOVE2. In addition, JHOVE2 introduces the concept of a *clump*, which is "a set of logically related files within a container or file set". This also applies to, for example, the file objects that constitute an ODF file.

In practice, JHOVE2 still identifies Microsoft Word documents as OLE2 objects; Open Document Text and EPUB files are identified as regular ZIP files. Also, one small (300 KB) EPUB file caused JHOVE2 to 'hang' (i.e. no exception occurred, but the processing of the file went on indefinitely and could only be stopped by manually interrupting JHOVE2).

HTML represents another class of composite objects, where, for example, an individual HTML file refers to external style sheets and images, which are all needed for proper rendering. The main difference with e.g. Open Document Format or EPUB is the absence of a physical container file. In JHOVE2 terms, the HTML example is simply a *clump* without a container file. However, the only comparable *clump* type that is supported by JHOVE2 at this stage is the Shapefile format (which is a GIS format that is based on a set of files that are not held together by any container).

8.16 User documentation

JHOVE2 is documented in a 39 page User's Guide in PDF format, which provides instructions on installing, configuring and using the software. The installation instructions in particular are very comprehensive. The User's Guide doesn't include a description (or even a mention) of JHOVE2's output format(s) or reported properties. Properties that are related to JHOVE2's feature extraction, validation and assessment functionalities are described in separate documents, but it would appear that the identification-specific output remains undocumented at this stage. The basic logic behind JHOVE2's output information is explained in a separate Architectural Overview document, but this is a piece of documentation that is more aimed at the level of developers than users. As a general remark, much of the currently available documentation on the JHOVE2 Wiki is very developer-oriented. Although the JHOVE2 team should be praised for taking developer documentation seriously, the currently available documentation will probably not satisfy the needs of JHOVE2's more general user audience.

8.17 Computational performance: one file at a time

8.17.1 KB Scientific Journals data set

The 'treeLaunch' tool was used to get an impression of Jhove2's performance when it is called to process one file at a time. It was set up to recursively traverse the KB Scientific Journals data set's directory tree, and run Jhove2 for each encountered file object. For an individual file object this results in a command line like:

```
jhove2 1-1-70.pdf
```

It took about 25 hours to analyse the whole KB Scientific Journals data set in this way. Figure 8-1 and Table 8-1 summarise the main results. On average JHOVE2 needs 7.6 seconds per file object, with a maximum of almost 20 seconds. This means that for the one-file-at-a-time scenario JHOVE2 is about 50 % faster than DROID 6, but compared to Fido or the Unix File utility this is still quite slow. Processing time appears to increase quite markedly with file size (Figure 8-2). One noteworthy feature of Figure 8-2 is that it shows some marked clustering around a number of narrowly-defined time intervals. These clusters can also be seen as minor 'peaks' in Figure 8-1, and they correspond to formats that have dedicated modules in JHOVE2 (mostly XML).

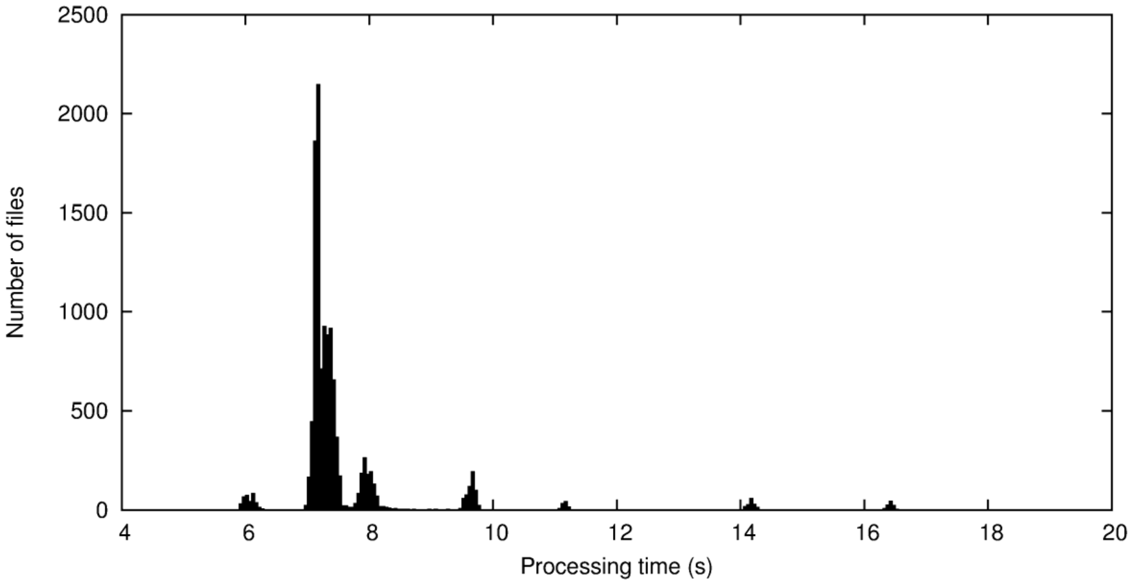

Figure 8-1 Distribution of processing time per file object for JHOVE2 one-file-at-a-time scenario, KB Scientific Journals data set.

Table 8-1 Summary performance statistics for JHOVE2 one-file-at-a-time scenario, KB Scientific Journals data set (expressed in seconds per file, except N). N=number of files; q1, median and q3 are 1st , 2nd and 3rd quantiles, respectively.

N	min	q1	median	mean	q3	max
11,892	5.906	7.156	7.266	7.610	7.422	19.704

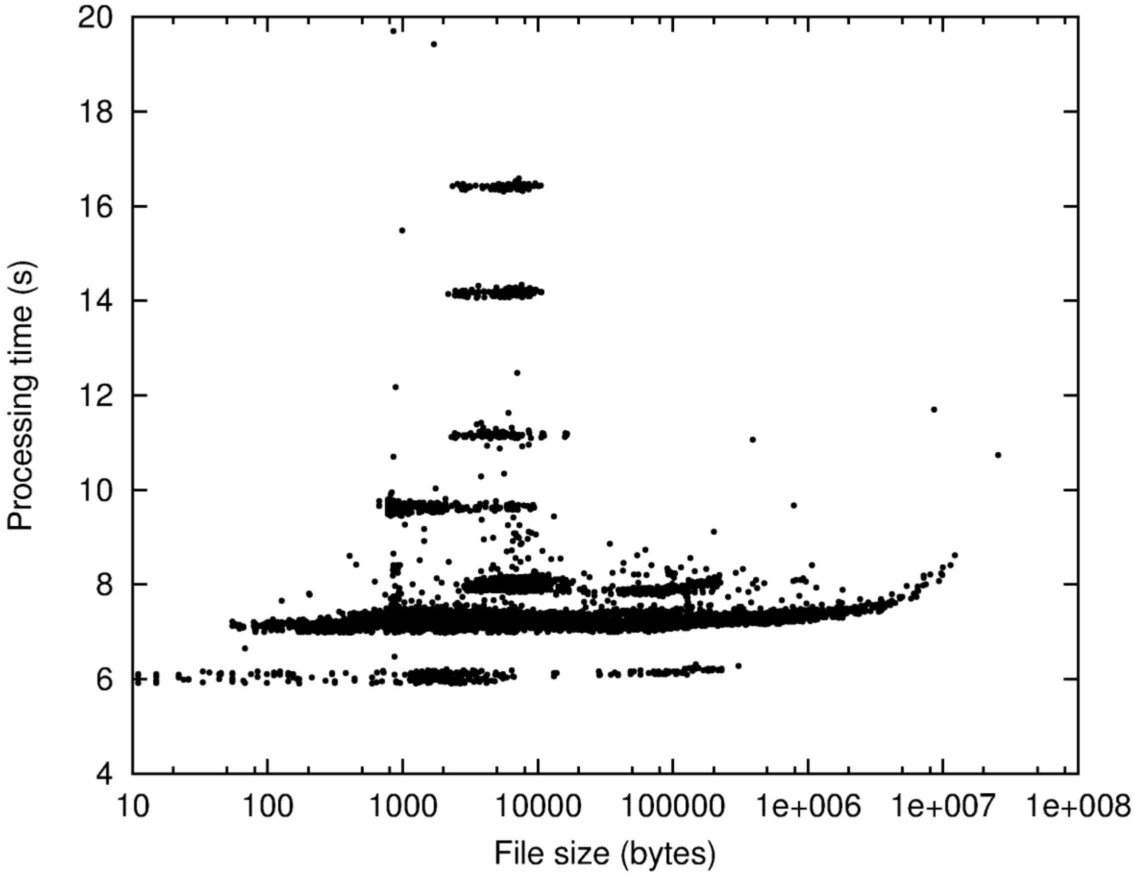

Figure 8-2 Scatter plot of processing time per file object versus file size for JHOVE2 one-file-at-a-time scenario, KB Scientific Journals data set. Note logarithmic scale on horizontal axis.

8.17.2 KB Large data set

Table 8-2 shows the results of an additional test that was done on the KB Large data set (again using the 'treeLaunch' tool). The table shows that processing time is strongly related to file size. Figure 8-3 illustrates this graphically. This is caused by the behaviour of DROID 4 (which is the identification tool that JHOVE2 uses 'under the hood'): DROID 4 always scans each file object in its entirety, which can make it very slow on large files[41]..

[41] See e.g. here: http://www.openplanetsfoundation.org/node/563 (comments section)

Table 8-2 Performance for individual files in KB Large data set

File name	Size (bytes)	Processing time (s)
dpo_tonal_00990.tiff	36,420,900	16.485
IMAGE000060_lossless_colour.jp2	49,818,414	17.406
IMAGE000060.TIF	114,664,502	23.86
SGD_19451955_0000002_ID371.pdf	325,101,508	66.689
DipAsset6984444754559678047.tar	534,968,320	100.799
KBDVD.iso	683,180,032	108.753
KBDVD17062011.img	693,993,472	101.392
DNEPABOstg.PST	1,895,515,136	190.847

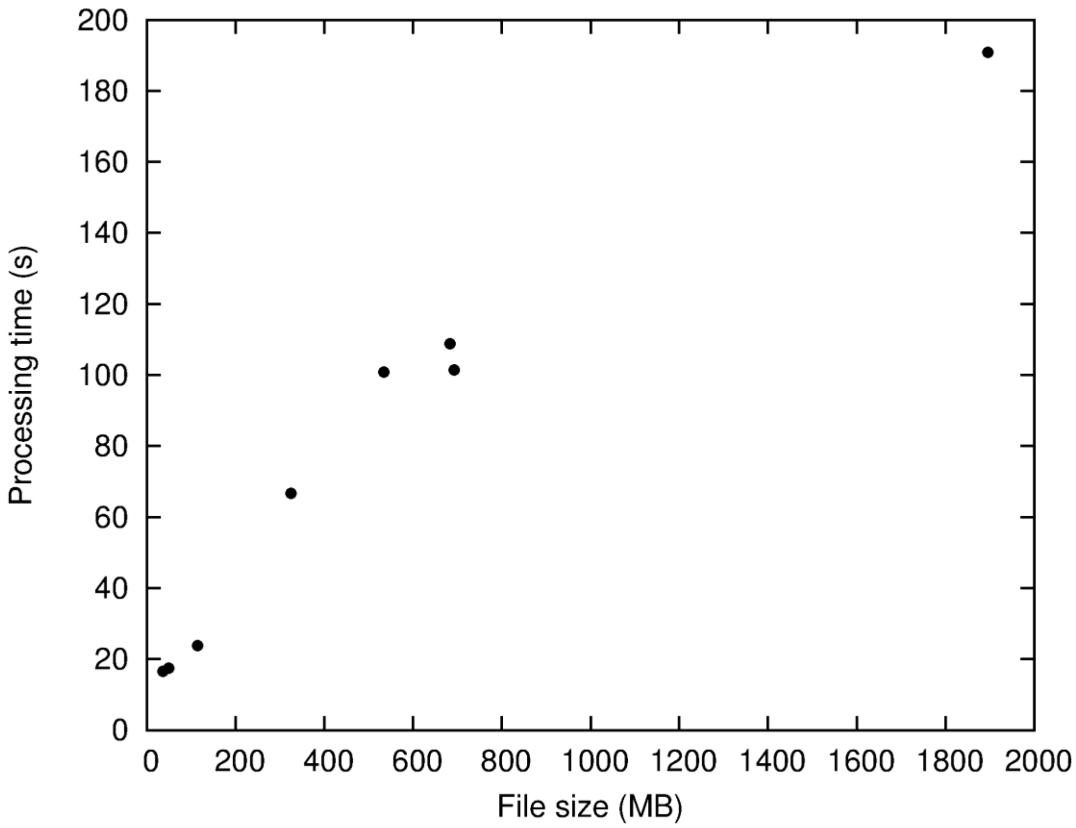

Figure 8-3 Scatter plot of processing time per file object versus file size for JHOVE2 one-file-at-a-time scenario, KB Large data set.

8.17.3 Additional tests on influence of file type and size

JHOVE2 is different from most of the other tools covered by this report in that it includes feature extraction and validation functionality for a limited number of file formats. One might expect that this would influence its performance, since the amount of processing is format-dependent. For instance, one would expect that formats that have a dedicated module (e.g.TIFF) would take more time than unsupported formats, as for the former JHOVE2 also performs feature extraction and validation. In order to get an indication of the impact of this on JHOVE2’s performance, we performed an additional test on a small set of 8 files. Performance was measured for two different JHOVE2 configurations:

- JHOVE2's standard configuration.
- A 'minimal' configuration, where all JHOVE2 modules apart from the "IdentifierCommand" were disabled[42].

Table 8-3 shows the results.

Table 8-3 Processing time for some format-size combinations using standard and minimal JHOVE2 configuration. Formats supported by a dedicated JHOVE2 module marked with *.

Format	**Size**	**Processing time (s), standard configuration**	**Processing time (s), 'minimal' configuration**
PDF	25 MB	10.4	9.0
PDF	69 KB	7.2	5.8
JPEG	69 KB	7.2	5.8
TIFF *	68 KB	8.1	5.8
TIFF *	110 MB	35.1	32.1
JP2	49 MB	15.6	14.2
ZIP *	8 MB	11.9	7.3
MS Word	2.5 MB	7.6	6.1

Two things are apparent from the table:

- Given a file of a particular size and format, processing time is not strongly related to whether or not the format has its dedicated JHOVE2 module.
- Performance is somewhat better for the 'minimal' JHOVE2 configuration, but the overall differences relative to the 'standard' configuration are comparatively small.

This all suggests that most of the processing time is consumed by DROID 4, and that JHOVE2's format-specific native modules are relatively fast. Since DROID 4 is known to be relatively slow, it may be possible to boost JHOVE2's performance significantly by using the latest version of DROID (6, see Chapter 4 of this report).

8.18 Computational performance: many files at a time

To test JHOVE2's performance while working on a large number of objects, we performed a recursive scan of all file objects in the directory structure of the KB Scientific Journals data set and measured the time needed for this. The following command line was used:

[42] This was done by commenting all other modules out in the JHOVE2 configuration file (config\spring\jhove2-framework-config.xml):

```
<property name="commands">
        <list value-type="org.jhove2.module.Command">
                <ref bean="IdentifierCommand"/>
                <!-- <ref bean="DispatcherCommand"/>
                <ref bean="DigesterCommand"/>
                <ref bean="AssessorCommand"/>
                <ref bean="AggrefierCommand"/> -->
        </list>
</property>
```

```
timeit jhove2.cmd D:\aipSamplesUnpacked -o D:\identOut\jhove2Out.txt
```

This gives the following result:

Elapsed Time:	0:35:42.541

So JHOVE2 needs about 35 minutes to scan 1.15 GB of data in the used test environment. For a total of 11892 file objects this corresponds to an average processing time of 0.18 seconds per file[43].

It is important to stress that these results are not directly comparable to DROID, Fido or the Unix File tool, since JHOVE2 also performs feature extraction and validation (for a limited number of formats). Since the scope of this report is restricted to identification, we did an additional test with the 'minimal' configuration that was mentioned in the previous section. This produced the following result:

Elapsed Time:	0:01:19.407

However, it turned out that the output in this case only gives contains identification results for the top-level directory ('D:\aipSamplesUnpacked'), which is identified as a 'directory'. All other subdirectories and files are listed in the output file, but it doesn't contain any identification information. This may seem surprising at first. However, JHOVE2 treats directories (and also file sets) as formats, which have dedicated format modules[44]. When JHOVE2 is instructed to scan a directory, it needs its 'directory' format module for processing the contents of that directory. However, *all* format modules are invoked through the 'dispatcher' command, and if the dispatcher is disabled (as is the case in the 'minimal' configuration), JHOVE2 will not get past the level of the root directory. This is not a bug: it is simply a consequence of JHOVE2's architecture.

8.18.1 Additional note on treatment of source units in JHOVE2

The above observations point to an important difference between JHOVE2 and most of the other tools that are addressed in this report. Basically, JHOVE2 *always* treats its (set of) command-line argument(s) as one single source unit (which may be either unitary or aggregate). So, when JHOVE2's command-line argument is a directory, it will treat its root as the principal 'parent' unit, and all its underlying sub-directories and files as 'child' units. This is different from, for example, analysing a directory in DROID 6, since DROID 6 will simply consider each file (and subdirectory) in that directory as an independent unit. Similarly, consider the following command line:

```
jhove2 rubbish.txt rubbish.pdf
```

One may expect that this will simply yield output related to files 'rubbish.pdf' and 'rubbish.txt'. Instead, at the highest level JHOVE2 treats this as a 'file set' (which is considered a format!) that contains 'rubbish.pdf' and 'rubbish.txt' as child units.

In many cases a user may simply want to process a collection of files as separate unitary source units (e.g. like in the 'old' JHOVE). This use case is currently not supported by JHOVE2, although it may be included in upcoming versions[45].

[43] On a side note, the size of the output file was about 191 MB.

[44] See also Page 39 ('How JHOVE2 Identifies Source Units') of the JHOVE2 User's Guide

[45] Source: e-mail Stephen Abrams, 2 august 2011 (JHOVE2-TECHTALK list)

8.19 Stability

8.19.1 JHOVE2 'hangs up' on EPUB/ZIP file

The analysis of one particular EPUB / ZIP file (size: 300 B) caused JHOVE2 to 'hang up': the software continued running for minutes without anything happening, and had to be interrupted manually.

8.19.2 Default location for writing memory objects

Although this is not strictly a stability issue, after installing JHOVE2 we initially experienced a problem where JHOVE2 would produce the following error message:

> 'Cannot initialize Berkeley DB environment',

followed by a succession of Java exceptions. This turned out to be a configuration issue: by default JHOVE2 uses (under MS Windows) the root of the C:\ drive for writing temporary memory objects. At the KB the root of C:\ is configured as read-only, which means that this will go wrong. This can be easily fixed by editing JHOVE2's configuration settings[46]. However, using the standard temporary file directory (as defined by Windows' TMP and/or TEMP environment variable) as a default would avoid this problem altogether (also, using the root of C:\ for writing temporary data is highly unusual, and we know of no other applications that do this).

8.19.3 JHOVE2 doesn't clean up its temporary files

Again not a stability issue, but problematic anyway: whenever JHOVE2 encounters a container unit (e.g. a ZIP file), it extracts its contents to the Java temporary directory (tmpdir) for further analysis. However, these files are not removed afterwards, which means that with time the Java temporary directory will grow indefinitely, which may lead to a variety of problems.

8.20 Error handling and reporting

Only very high level error messages are written to the console's standard error device. However, all other error messages, warnings and informative messages are included in JHOVE2's output files (which are always written to standard out).

8.21 Provision of event information

The event information that is provided by JHOVE2 is extremely elaborate. It includes detailed information on the environment in which JHOVE2 was run, and detailed information on individual JHOVE2 modules. In fact, most of the information in a JHOVE2 output file is event information. The sheer amount of event information can even be a little overwhelming. This problem should go away once style sheets are available for transforming JHOVE2 output files to something more digestible (e.g. PREMIS).

8.22 Maturity and development stage

JHOVE2 is still a very young tool. So far two prototypes have been released (August 2009 and April 2010, respectively), followed by a 'first production release' in April 2011. This is also the version that is evaluated here. The term 'production release' suggests stable software that is ready to be used in

[46] This is done by changing the property "envHome" in JHOVE2's memory management configuration file (explained on pages 22/23 of the User's Guide).

an operational setting. This is somewhat misleading, as JHOVE2 in its current form does not appear to be ready for operational use yet[47].

8.23 Development activity

The development of JHOVE2 has been very active since the start of the project in 2008. Also, several third-parties that are not part of the JHOVE2 project team have started developing JHOVE2 modules. An example is the NetCDF module that is being developed by the Alfred Wegener Institute for Polar and Marine Research. Maintenance and enhancement activity is funded by the three partner institutions in the JHOVE2 project team, which makes it slightly unclear at this stage how (and at what rate) further development will continue.

8.24 Existing experience

Given that it is still a very new tool, experience with JHOVE2 is still very limited.

8.25 Unidentified files

Not investigated (most likely similar to DROID).

8.26 Conclusions

The evaluation of JHOVE2 revealed a number of specific strengths and weaknesses.

JHOVE2's main strengths appear to be:

- JHOVE2 offers an integrated approach for file characterisation that includes identification, feature extraction, validation and policy-based assessment.
- JHOVE2's modular architecture is a particular strength. It makes it attractive for third parties to contribute to JHOVE2 by developing new modules. Also, it makes it possible to wrap external tools in the JHOVE2 framework.
- JHOVE2 provides extremely comprehensive output, including detailed event information. Currently this makes the JHOVE2 output rather difficult to digest, but once the style sheets for transforming JHOVE2's intermediate output format to METS and PREMIS are in place (which will be included in the next release) this could turn into a major advantage over other existing tools.

Potential problems for use in preservation workflows are:

- JHOVE2 is shipped with a very old version of the DROID signature file, which also contains undocumented edits. It is not clear what happens if a user replaces the default signature file with a more recent one, which raises some concerns on extendibility.
- The User's Guide doesn't provide any information on either JHOVE2's output format or its reported properties. For the feature extraction, validation and assessment modules this is partially covered by separate documents (which were not reviewed here), but it seems that the identification-specific output remains undocumented at this stage. This may not be a problem once style sheets to convert to METS or PREMIS are in place, but these are not included with the current release.

[47] Reasons for this include the current use of an undocumented, complex intermediate output format; the lack of any style sheets to transform the output to something more usable, and the fact that some format modules (PDF, JPEG 2000) are still missing.

- Much of JHOVE2's documentation is very developer-oriented at this stage. This may scare potential users of the software away. This could be remedied by improved and more elaborate user documentation.
- The current development stage of JHOVE2 is unclear. The evaluated version of the software is announced as a production release, but the fact that JHOVE2 is currently only able to produce output in a complex, intermediate output format which is also completely undocumented would by itself make the use of the software in an operational setting practically impossible. Moreover, current the lack of some format modules, style sheets for output transformations and a more user-oriented documentation all suggest that the software is currently still in beta stage. By itself this is not a problem, but the developers should be clear about this. Presenting JHOVE2 as a production tool at this stage may confuse its potential user base, which could be counter-productive in the long run.
- Although JHOVE2's data model is particularly suited for dealing with composite objects, on the identification side this doesn't work out too well in practice because of its dependency on DROID 4. This means that for MS Office documents, Open Document files and EPUB files only the container format (OLE2, ZIP) is identified. A move towards DROID 6 would improve this.
- The dependency on DROID 4 results in a relatively poor computational performance, especially for large file objects, and when JHOVE2 is invoked for one file object at a time. Also here, a move towards DROID 6 would improve this.
- JHOVE2 can be used for many files at a time scenarios (e.g. processing a directory tree), but in that case all objects and subfolders will be interpreted as child units of the root directory. Whether this is a problem or not depends on the context; moreover the currently used output format (see above) makes it difficult to make any assessment of this.
- In principle it is possible to configure JHOVE2 to do identification only, and skip any additional processing. The (mainly architectural) choice to treat directory trees and file sets as formats limits these possibilities in practice, since the identification of file objects inside folders still requires the invocation of format modules.
- The tests revealed a number of issues: the default location for writing temporary memory objects; the creation of temporary that are not cleaned up, and the Windows launcher scripts that only work from their installation directory. These are all relatively minor and (most likely) easy to solve.

Similarly to the evaluation of FITS, it is important to stress that the results of JHOVE2 cannot be directly compared to those of the other tools in this report. The main difference is that identification is only one part of JHOVE2's functionality: it also includes feature extraction, validation and policy-based assessment. These are all outside of the scope of this evaluation. It also means that any computational performance results cannot be directly compared with dedicated identification tools (although JHOVE2's performance issues appear to be caused mainly by DROID 4, with JHOVE2's native modules adding very little overhead).

Based on the above considerations, if the scope is limited to identification only, JHOVE2 does not appear to be the most obvious choice for inclusion in the SCAPE architecture. If the scope is widened to include feature extraction, validation and/or policy-based assessment the conclusion may be very different.

9 Concluding observations and suggestions for further work

The previous chapters showed that each of the evaluated tools has their specific strengths and weaknesses. We will not repeat these here. However, there are a couple of themes and issues that are common to many of these tools, and these may show a way to how SCAPE could contribute to the improvement of identification tools.

9.1 Performance of Java-based tools

For all the evaluated tools that are written in Java (DROID 6, FITS JHOVE2), performance is problematic for the 'one file at a time' use case. This is primarily caused by the slow initialisation of Java applications. The main problem here is that a typical 'identification' operation on a single file object is very fast (order of magnitude: milliseconds), but that this is not worth much if an application first needs several seconds to initialise. This also explains why a tool such as Fido, which is written in Python, outperforms the Java-based tools by several orders of magnitude for this use case. Even though Python is actually a 'slower' language than Java, the absence of any significant initialisation penalty ultimately results in a much better performance. The results of the DROID and Fido evaluations also demonstrate that these performance differences disappear for the 'many files at a time' use case (with DROID being marginally faster). A fairly obvious solution for improving the performance for 'one file at a time' use cases would be to invoke the tools through the Java API, but this only works if the 'calling' workflow management system is also Java based. The question is: is this a restriction that is acceptable for the designated user community of these tools? There appear to be two important considerations here:

1. The number of file objects per tool invocation: one invocation for each individual file versus one invocation for multiple files
2. The interface through which the tool is invoked: command-line interface versus Java API.

Table 9.1 below gives a schematic overview of the implications of the above on performance. The combination of 'one invocation for each individual file' and an invocation through the command-line interface is particularly problematic. Should the majority of (potential) users want to deploy these tools in this way, then this would imply a mismatch between the tools and the requirements of their user base. Whether this is actually the case is a completely different matter, but it is a consideration that has major implications for operational workflows.

Table 9-1 Performance of Java-based tools as a function of number of file objects per invocation and invocation method ('-' indicates poor performance, '+' indicates good performance)

	One invocation - one file	One invocation – multiple files
Command line interface	-	+
Java API	+	+

9.2 Identification of text-based formats

Although the current analysis doesn't directly address identification accuracy, a recurring observation that applies to all of the investigated tools is that the identification of text-based formats (including XML) is problematic. This is largely a limitation of signature-based identification, which works well for most binary formats, but is less suited to text-based formats. The identification of such formats could be improved significantly by using alternative identification methods that do not rely on signatures. Some examples are:

- The identification of XML could be improved using standard XML parsers that are available for all modern programming languages. Some ideas and suggestions are given in Blekinge (2011) and van der Knijff (2011).
- Some text-based formats can be identified by presence of specific keywords. This applies to (but is not limited to) nearly all programming and scripting languages. The results of the current analysis show that text files that contain source code or scripts often remain unidentified. A possible solution may be to identify such files using statistical techniques such as Bayesian spam filtering[48].

Both methods can be implemented using readily available software libraries, and the resulting tool(s) could be complementarily to signature-based tools.

9.3 Extensions to Unix File?

An earlier study by Underwood (2009) discussed several extensions to the Unix File utility that would make this tool better suited to use in digital archives. These are mostly related to the creation and management of file signatures. Since The Unix File utility is a mature tool that is widely used, supports a wide range of file formats and has an outstanding computational performance, improving the remaining problem areas could be an interesting option to improve and streamline signature-based identification in archival settings.

[48] See for instance the discussion here: http://stackoverflow.com/questions/475033/detecting-programming-language-from-a-snippet

Acknowledgements

We would like to thank everyone who has provided feedback on earlier drafts of this document. The input from the following people has been particularly helpful : Andrew Fetherston (The National Archives), Maurice de Rooij (National Archives of the Netherlands; Open Planets Foundation), Andrea Goethals, Spencer McEwen (Harvard University Library), Stephen Abrams (California Digital Library), Sheila Morrissey (Portico), Miguel Ferreira (KEEP Solutions), René Voorburg, Judith Rog, Bart Kiers, Barbara Sierman (KB/National Library of the Netherlands), Bram Lohman (Tessella).

References

Blekinge, A. A new direction in file characterisation. Blog, Open Planets Foundation, 2011. Link: http://www.openplanetsfoundation.org/blogs/2011-02-17-new-direction-file-characterisation (accessed 20 September 2011).

DROID 6 Help. The National Archives, 2011.

FITS User Guide. Link: http://code.google.com/p/fits/wiki/user_guide (accessed 20 September 2011).

JHOVE2 User's Guide. JHOVE2 Project Team, 2011.

JHOVE2 Functional Requirements 1.4. JHOVE2 Project Team, 2009.

Underwood, W. Extensions of the UNIX File Command and Magic File for File Type Identification. Georgia Tech Institute of Technology, 2009.

Van der Knijff, J., Blekinge, A. & Schlarb, S. WP 9: evaluation framework for characterisation tools. Internal report, SCAPE project, 2011a.

Van der Knijff, J., Blekinge, A. & Schlarb, S. WP 9: target characterisation tools. Internal report, SCAPE project, 2011b.

Van der Knijff, J. Improved identification of XML: a Python experiment. Blog, Open Planets Foundation, 2011. Link: http://www.openplanetsfoundation.org/blogs/2011-07-11-improved-identification-xml-python-experiment (accessed 20 September 2011).

.