View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Development of modules for the GNU PDF project

Albert Meroño Peñuela

June 22, 2011

To my immortal beloved

Acknowledgements

It is a pleasure to thank those who made this Final Degree Project possible. In first place, I owe my deepest gratitude to my project director, Antoni Soto, who advised me with invaluable suggestions and always had time for another meeting or one last revision. Thank you very, very much. I am also indebted to many GNU PDF members I have met during this time, but I would like to specially thank José E. Marchesi for that great meeting in Barcelona, illustrative discussions and guidance in the paths of software freedom, and Aleksander Morgado for his great hacks and deep source reviews. Free Software is a reality thanks to people like you. I would also like to show my gratitude to many other FIB professors who helped me in this process, specially José Miguel Rivero. All my gratitude for all my colleagues in the IDT, who kindly allowed me to develop part of this project during work hours. You are too many to mention, so I would like to sincerely thank you all.

I would like to specially thank the most important persons in my life, for the patience, dedication and support that they have given to me all these years. So thank you mum, Júlia, thank you dad, Dídac, and thank you sis, Mariona, for everything. I want to thank all my friends, who know I owe them many hours. Thank you Nayef, thank you Alex, thank you Pere, and thank you Jose. And thanks to you, Ingrid, for so many, many things I just can not explain. Thank you for loving me.

Barcelona, June 2011

Contents

1 Introduction **9**
- 1.1 Motivation . . . 9
- 1.2 Project overview . . . 10
- 1.3 Goals . . . 12
  - 1.3.1 General goals . . . 12
  - 1.3.2 Specific goals . . . 12
- 1.4 Report organisation . . . 12

2 Entering GNU PDF **13**
- 2.1 Introduction . . . 13
- 2.2 Brief history of GNU PDF . . . 14
- 2.3 Joining GNU PDF . . . 16
  - 2.3.1 GNU PDF Information for Newcomers . . . 16
  - 2.3.2 `#pdf` channel at `freenode.net` . . . 17
  - 2.3.3 Entering the mailing list . . . 17
- 2.4 Licensing . . . 17
  - 2.4.1 Copyright assignment . . . 18
  - 2.4.2 University disclaimer . . . 18
- 2.5 Project methodology and organisation . . . 18
  - 2.5.1 Library architecture . . . 18
  - 2.5.2 Sources retrieval and GNU PDF installation . . . 21
  - 2.5.3 The development branch . . . 22
  - 2.5.4 Iterations, patch proposals and review process . . . 23
  - 2.5.5 Tasks management . . . 24
  - 2.5.6 Library quality . . . 25
  - 2.5.7 Tools . . . 27
- 2.6 The first task choice . . . 28

3 The UUID module **29**
- 3.1 Universally unique identifiers . . . 29
  - 3.1.1 Internal structure . . . 30
- 3.2 Iterative development . . . 31
- 3.3 First iteration . . . 31
  - 3.3.1 Requirements analysis . . . 31
    - Requirements sources . . . 31
    - Functional requirements . . . 32

Non-funtional requirements . . . 33
3.3.2 Specification and Design . . . 36
Existing, suitable UUID implementations . . . 36
Design decisions . . . 39
3.3.3 Implementation . . . 41
Introducing dependency with *libuuid* . . . 42
UUID Module coding . . . 45
3.3.4 Testing . . . 46
3.3.5 First patch . . . 47
3.3.6 Review . . . 48
3.4 Second iteration . . . 48
3.4.1 Implementation . . . 49
UUID Module coding . . . 49
3.4.2 Testing . . . 51
3.4.3 Second patch . . . 51
3.4.4 Review . . . 51
3.5 Third iteration . . . 52
3.5.1 Implementation . . . 53
UUID Module coding . . . 53
3.5.2 Testing . . . 55
3.5.3 Third patch . . . 55
3.5.4 Review . . . 55
3.6 Fourth iteration . . . 55
3.6.1 GNU PDF Library Testing Infrastructure . . . 56
3.6.2 Testing . . . 57
`pdf-types-uuid-generate.c` . . . 57
`pdf-types-uuid-string.c` . . . 57
`pdf-types-uuid-equal-p.c` . . . 59
3.6.3 Testing environment execution . . . 59
3.6.4 Fourth patch . . . 60
3.6.5 Review . . . 60

4 PDF standards requirements **61**
4.1 Introduction . . . 61
4.2 Concept glossary . . . 62
4.3 PDF standards . . . 63
4.3.1 Standardized subsets of PDF . . . 64
PDF/X . . . 64
PDF/A . . . 65
PDF/E . . . 65
PDF/VT . . . 65
4.3.2 Non-standardized subsets of PDF . . . 66
PDF/UA . . . 66
PDF/H . . . 66
4.4 PDF/A vs PDF 1.4 requirements . . . 66
4.4.1 File structure . . . 67
4.4.2 Graphics . . . 67

4.4.3 Fonts . . . 68
4.4.4 Transparency . . . 68
4.4.5 Annotations . . . 69
4.4.6 Actions . . . 69
4.4.7 Metadata . . . 69
4.4.8 Logical structure . . . 69
4.4.9 Interactive Forms . . . 70
4.5 A Conformance Module proposal . . . 70
4.5.1 Conformance Module API . . . 71
Requirements Management . . . 71
Conformance Context Management . . . 73
Conformance-Requirements Mapping Management . . . 74

5 A PDF object parser proposal **78**
5.1 Introduction . . . 78
5.2 PDF objects . . . 81
5.2.1 Boolean objects . . . 81
5.2.2 Numeric objects . . . 81
5.2.3 String objects . . . 82
5.2.4 Name objects . . . 82
5.2.5 Array objects . . . 83
5.2.6 Dictionary objects . . . 83
5.2.7 Stream objects . . . 83
5.2.8 Null object . . . 84
5.2.9 Indirect objects . . . 84
5.3 A grammar for PDF objects . . . 85
5.3.1 First proposal, G_1 . . . 86
5.3.2 Second proposal, G_2 . . . 89
5.3.3 G_1 and G_2 pros and cons . . . 91
5.4 Parser development . . . 92
5.4.1 Lookahead . . . 92
5.4.2 Parsing strategy . . . 93
Generic parsing algorithm . . . 94
Bottom-up strategies . . . 94
Top-down strategies . . . 95
5.4.3 Requirements . . . 95
Goal . . . 95
Functional requirements . . . 95
Non-functional requirements . . . 96
5.4.4 Specification . . . 96
Usage . . . 96
5.4.5 Design . . . 97
Parsing window . . . 97
Flags . . . 97
Parsing schema . . . 97
5.4.6 Implementation . . . 98
`rec-pred-parse.h` . . . 98

rec-pred-parse.c . . . 100
5.4.7 Testing . . . 102

6 Documentation **104**
6.1 Source code documentation . . . 104
6.2 gnupdf.texi . . . 104
6.3 A GNU PDF developer's blog . . . 105
6.4 Social Networks . . . 105
6.5 GNU PDF Knowledge Database . . . 105
6.6 Wikipedia contributions . . . 106
6.7 IRC community . . . 107

7 Budget and execution **108**
7.1 Budget . . . 108
7.1.1 Software costs . . . 108
7.1.2 Hardware costs . . . 108
7.1.3 Staff costs . . . 108
7.1.4 Final cost . . . 108
7.2 Execution . . . 109
7.2.1 UUID module . . . 109
7.2.2 Conformance tasks . . . 110
7.2.3 Parser development . . . 110
7.2.4 Report and documentation . . . 111
7.2.5 Overall . . . 111

8 Conclusions **113**
8.1 Goals achieving . . . 113
8.2 Further work . . . 115
8.3 Closing remarks . . . 116

A FSF licensing **124**

B Licensing notes of UUID libraries **130**
B.1 libbuid in e2fsprogs . . . 130
B.2 OSSP uuid . . . 131

C PDF/A vs PDF 1.4 requirements **132**
C.1 File structure . . . 133
C.2 Graphics . . . 134
C.3 Fonts . . . 137
C.4 Transparency . . . 139
C.5 Annotations . . . 139
C.6 Actions . . . 140
C.7 Metadata . . . 141
C.8 Logical structure . . . 143
C.9 Interactive Forms . . . 146

D ANTLR prototypes **147**
D.1 EBNF grammars 147
D.1.1 G_1 147
D.1.2 G_2 148

List of Figures

2.1 Layered architecture of the GNU PDF Library 19
2.2 Architecture of the Base Layer 20
2.3 Source tree directory of the GNU PDF Library 23
2.4 Tasks workflow 24
2.5 *Flyspray* task management system, overview 25
2.6 Cyclomatic Complexity Report 26
2.7 API Consistency Report 26
2.8 Unit Testing Report 27

3.1 Base layer architecture 30
3.2 *Flyspray* task management system, task details 32
3.3 Details of task #122, UUID Module 33
3.4 Required data types for the UUID Module 33
3.5 First required function for the UUID Module 34
3.5 Second required function for the UUID Module 34
3.5 Third required function for the UUID Module 35
3.6 Retrieval of source code for candidate UUID libraries 36
3.7 Requirements e-mail to Theodore Ts'o 37
3.8 Reply e-mail from Theodore Ts'o 38
3.9 Jose E. Marchesi's points on libuuid requirements 39
3.10 The *libuuid* API 40
3.11 First approach to an UUID module API 41
3.12 Flow diagram of configure, autoconf and automake 43
3.13 Changes in configure.ac 44
3.14 Changes in `src/Makefile.am` 44
3.15 Question about name-based UUIDs convenience 46
3.16 Reply about name-based UUIDs convenience 47
3.17 First version of the UUID data type 48
3.18 Comments on first UUID module patch (1/2) 48
3.19 Comments on first UUID module patch (2/2) 49
3.20 Second version of the UUID data type 50
3.21 Comments on second UUID module patch 52
3.22 Detail of extra parameters for `pdf_uuid_string` 53
3.23 Detail of UUID ASCII size constant 53
3.24 Third patch acceptance notification 55
3.25 Library testing strategy 56

3.26 Unit testing architecture 57
3.27 Unit testing sources 58

5.1 Phases of a compiler 79
5.2 Indirect object rule for G_1 87
5.3 Array object rule for G_1 87
5.4 Array / dictionary value rule for G_1 88
5.5 Dictionary object rule for G_1 88
5.6 Key-value pair rule for G_1 89
5.7 Stream object rule for G_1 89
5.8 Indirect object rule for G_2 89
5.9 Contained object rule for G_2 89
5.10 Atomic object rule for G_2 90
5.11 Array object rule for G_2 90
5.12 Array or dictionary value rule for G_2 90
5.13 Dictionary object rule for G_2 91
5.14 Key-value pair rule for G_2 91
5.15 Stream object rule for G_2 91

6.1 Portable Document Features visits 105
6.2 Portable Document Features countries visits 106
6.3 PDF Standards article visitors 106

7.1 Gantt chart of the project 109
7.2 Gantt chart of the UUID module 110
7.3 Gantt chart of the conformance tasks 110
7.4 Gantt chart of the parser development 111
7.5 Gantt chart of report and documentation writing 111
7.6 Chart of working hours used during the FDP 112

8.1 The GNU PDF contributing methodology 117

A.1 Envelope of the FSF copyright assignment letter 124
A.2 Contents of the FSF copyright assignment letter 125
A.3 FSF copyright assignment (1/2) 126
A.4 FSF copyright assignment (2/2) 127
A.5 FSF university disclaimer (1/2) 128
A.6 FSF university disclaimer (2/2) 129

List of Tables

3.1 The fields of a UUID 30
3.2 Mapping between *libuuid*'s API and UUID module API 41

4.1 The ISO 15930 (PDF/X) series and conformance levels 64
4.2 The ISO 19005 (PDF/A) series and conformance levels 65

5.1 Correspondence between tokens and input string chunks 86
5.2 Lookaheads and parsing window example 94

7.1 Budget for hardware costs 108
7.2 Budget for staff costs 109
7.3 Budget for the project 109
7.4 Time spent by task 112

Chapter 1

Introduction

This Final Degree Project (FDP) is a collaboration with a Free Software project: **GNU PDF**. This report describes the work performed during this collaboration.

The goal of the GNU PDF project is to develop and provide a *free*, *high-quality*, *complete* and *portable* set of libraries and programs to manage the PDF file format, and associated technologies [31].

Currently, the main activity in the GNU PDF project is the development of the GNU PDF Library. This library provides functions to read and write PDF documents conforming to standardized specifications.

The main goal of this FDP is the development of modules for the GNU PDF Library. As described down below, the current status of the development aims at tasks involved in lexical, syntactic and semantic analysis of PDF files. These processes begin in the object layer of the library. Additionally, some other tasks on implementing basic features are still pending. These features belong to the base layer. This collaboration FDP makes contributions to both layers.

In this introductory chapter an overview of this FDP motivation, goals and organisation is presented.

1.1 Motivation

This project was born on September 2010. It has its roots in the *High Priority Free Software Projects* webpage [34]. There, a number of Free Software projects in need of support from the Free Software community is listed. The same webpage states that:

> Our list helps guide volunteers and supporters to projects where their skills can be utilized, whether they be in coding, graphic design, writing, or activism.

At that time, the project listed in the top position was GNU PDF. This was part of the motivation: having the chance to support and collaborate with a project that the Free Software Foundation tagged as *high priority* was very motivating by itself. However, the rest came from the current situation regarding the PDF format and current tools, free or not, to manage it, which is a complex situation. The same webpage [34] tries to summarize it to catch volunteers' attention:

> The PDF format is an international standard (ISO 32000) and current free software support for PDF contains few of the supported features. We believe that we urgently need a collection of free software PDF libraries and programs that can fully implement this standard and provide users with many of the features they are currently missing in their PDF applications, such as support for interactive forms and JavaScript validation, annotation support, and embedded movies and 3D artwork—just to name a few.

This is absolutely true: current free software tools for managing PDF documents do not support ISO 32000 features. This has a direct consequence: final users do not have a free replacement for PDF privative software (provided by Adobe Systems, Inc., who created the PDF format in 1991), which really supports all the standard features. This fact prompts two major problems:

1. Final users that do not want to use privative software to perform their daily PDF tasks are missing important PDF capabilities. This situation is currently unsolved: free software users have to move to privative software to be able to use these functionalities (Adobe Acrobat X Pro has an approximate price of US $449.00); or, even worse, they have to work giving up them.

2. Apart from particular users, the whole digital document industry is suffering because the lack of a free PDF replacement for libraries and tools. Moreover, businesses are enforced to pay for the Adobe SDK in order to develop standard conformant PDF applications. This conformance is brought by the Adobe SDK, because today is the only PDF library that is certified on the most commonly used PDF formats (PDF 1.4, PDF 1.5, PDF 1.6, PDF 1.7 and the subsets PDF/A, PDF/X and PDF/E). Without a free software certified library, Adobe SDK will continue to be the only choice for the PDF digitial document industry, as well as the public sector and the rest of industries and final users.

Currently there exist a number of free PDF tools and libraries, implementing basic PDF reading and writing. However, these tools were developed essentially before the release of the ISO 32000-1:2008 standard. This has two critical consequences in both features (because standardized capabilities like annotation support or JavaScript form validation were previously unpublished) and conformance (because these tools are not written with the conformance problem in mind at all). As this is a design problem, improving these tools to make them conformance-aware may imply higher developing efforts than taking a from-scratch approach. This is the reason for the GNU PDF project to centralize a PDF conformant library.

1.2 Project overview

When the project started, it was quite difficult to establish a well defined set of goals. It was found that, before establishing these goals, it was necessary to enter the GNU PDF community, interact with its members, collect uncompleted or programmed tasks still to do, and study the overall development status of the GNU PDF Library.

The results of this first work can be found in chapter 2. In brief, the library was in an early stage of development. The documentation showed that it was organised in a four-tier

architecture, consisting of a base layer, an object layer, a document layer, and a page contents layer. This stack suggested a bottom-up approach, beginning with the base layer and following with the object layer and the rest of layers. Specification, design, implementation and test stages were very advanced in the base layer. This layer supports basic modules for very low level functionalities that the library has to provide, like basic types, memory management, file system access, filtered streams, encryption and tokenisation, to name a few. However, the rest of layers were in a very early stage of development (like the object layer) or were not even started (like document and page contents layers). At this point, it was clear that the collaboration of this FDP would develop very low-level contributions, as the volunteers' effort was concentrated on finishing modules of the base layer and beginning with the first tasks in the object layer.

The first step for getting involved was to send a presentation message to the `pdf-devel` mailing list, in order to notice the GNU PDF community that a new contributor was joining in the context of this FDP. The community replied with a kind welcome, and suggested to take any `NEXT` task of the task pool (see chapter 2 for details). The choice of this task, as well as the reasons that lead to choose it, are detailed below, in chapter 2. In summary, it can be stated for now that this task was required not to be very intensive in quantitative terms of *lines of code*. Instead of that, it was required to help to acquire knowledge about how the library was structured, how the GNU PDF development dynamics run, and how the software development methodology works. Hence, submitting a high amount of source code, or developing a significant number of functionalities, was considered non relevant for this first approach. The pool was well examined, and the task that better fit these and other requirements was found to be one named *Development of the UUID module*. All the work developed around it is described in chapter 3.

At the same time, some private messages with the main developer and maintainer of the GNU PDF project, José E. Marchesi, followed. Apart from these messages, a meeting was arranged in Barcelona. As a result of those interactions, some key issues regarding standard and specification conformance, as well as important compiler needs that the library was lacking, were discussed. It was agreed to establish a framework for this FDP that did not follow the standard developing procedure (which is strongly focused on the task cycle detailed in chapter 2). This framework would allow the FDP to develop proposals on both conformance and compiler tasks without following the usual methodology.

At this point, it was clear that the collaboration project would cover the following items:

1. The development of the UUID module, under the strict development methodology of the GNU free software projects, and particularly the development methodology of the GNU PDF project. This consists of adding improvements comming from contributors to the current development branch with a patch system.

2. The development of studies, comparisons, conclusions and proposals regarding the library conformance with respect to official standards and PDF language specifications. This task would not follow the standard patch submitting procedure.

3. Setting up a proposal for a syntax checking algorithm for PDF files (that is, a parser for PDF files). Again, this task would not follow the standard patch submitting procedure.

The specific goals of the project are detailed in section 1.3.

1.3 Goals

This section describes the goals of this FDP. To do so, first a list of general goals is presented, and then an enumeration of more specific goals is detailed.

1.3.1 General goals

The general goals of this project are the following:

1. To learn how to collaborate and work with and for the community of a Free Software project.
2. To contribute the GNU PDF Library with source code, improving its functionalities and adding more of them.
3. To help the GNU PDF community with knowledge, reports, articles and useful information about PDF technologies.
4. To spread the GNU PDF initiative across the world.

1.3.2 Specific goals

The specific goals of this project are the following:

1. To begin a long term contribution with the Free Software community through this FDP collaboration in the GNU PDF project.
2. To develop the UUID module, a still non-implemented module in the base layer.
3. To study the PDF standard and specification domain.
4. To provide a roadmap on formal comparison of PDF requirements found in PDF standards and specifications.
5. To provide a specification proposal of a conformance module, probably being located in the base layer.
6. To develop a PDF object parser, choosing the most appropriate parsing algorithm based on a suitable PDF object grammar.
7. To contribute the GNU PDF project documentation.

1.4 Report organisation

This report is organised as follows. First, chapter 2 describes the process of joining, meeting and getting invloved with the GNU PDF project. Second, chapter 3 reports the development process of the UUID module, which was needed in the base layer. Third, chapter 4 explains the importance of standards for the GNU PDF project, and gathers all tasks performed regarding analysis of requirements in PDF standards and specifications. Fourth, chapter 5 describes the tasks involving the development of a PDF object parser. Fifth, chapter 6 shows all documentation tasks, including diffusion efforts. Sixth, chapter 7 describes the FDP budget and execution. Finally, chapter 8 points some important conclusions and further work.

Chapter 2

Entering GNU PDF

This chapter describes the process of joining a Free Software project community, focusing on the particular case of GNU PDF, and explains the most important characteristics of its development group, its internal dynamics and available tools for helping volunteers to get organised. A brief description of the process of joining as a volunteer is also offered.

2.1 Introduction

Engaging in a Free Software project always come with social interaction with the rest of the development community. This does not happen in the Free Software planet only: human interaction is the first non-strictly programming challenge a programmer must face when he or she is working in a team. All medium-large projects require, to a greater or lesser extent, the coordination of a human team to responsibly assign resources, plan tasks, establish workflows and deal with conflicts.

The first interaction of this class occurs often at University, usualy in the second year of computing studies [9]. Students are challenged to develop a project in working teams of two or three programmers, facing the issues invloved in team software developing. Later, once studies have been finished, these skills are put to the test, using large scale software projects involving not only additional programmers, but a full set of different stakeholders that have critical consequences on the activity of developing software.

The team work in Free Software development has some of these facets, as well as the lack of some others and the exclusive of some particular ones. For instance, usually Free Software is not influenced by a strict budget or completion times, and lacks the presence of the customer role. This is because Free Software is not a product to be supplied to a client in order to satisfy some customer need; the motivations, driving forces and business engine for Free Software is something more related with ethics, social responsibility, and recognition from peers [45]. However, Free Software projects usually try to satisfy some computing need that a significant part of society demands.

The chapter is organised as follows. First, a brief history of the GNU PDF project is provided. Second, the process of joining the GNU PDF project is described. Third, some licensing issues concerning this FDP, as well as licensing issues concerning any developer who wants to contribute, are depicted. Fourth, the project methodology and organisation is shown. Fifth, basic tools for managing oneself while contributing are detailed. Finally, the process of choosing the first task to contribute is analysed.

2.2 Brief history of GNU PDF

The history of the GNU PDF project begins with the José E. Marchesi, who was maintaining GNU gv and GNU ghostscript and was interested in the ghostscript's PDF interpreter. He realised then that PDF support in Free Software was not covering users needs. Some of those needs are documented in the GNU PDF Knowledge Database [22]. For instance, on one hand interactive forms were (and are still) unsupported in PDF free software libraries. On the other hand, public institutions use commonly these forms in PDF documents that citizens usually have to fill. This was leading to a implicit enforcing to citizens to rely on privative software while filling these forms: no free alternative existed to Adobe's tools.

Free Software community would be able to complain about this, but in 2008 the ISO published the first full specification standard of PDF: ISO 32000-1:2008 [2]. This event cancelled the protest: nobody was then preventing the Free Software community to implement its own fully functional and compliant PDF library.

At that time existed, of course, several free software packages covering PDF format support. However, the option of improving these packages to make them also conformant with recently published ISO 32000-1:2008 (moreover, to make them conformant with all standardized subsets of PDF) was refused. The following public communication between José E. Marchesi and Raph Giles describes very well the foundings of GNU PDF:

> *The "Goals and motivations" page has lots of information about why you'd want a PDF implementation, but nothing about why you're writing a new one, or what your goals are. There's already Ghostscript (GPLv2) and libpoppler/xpdf (GPLv2+) on the rendering side, and libcairo (LGPL2.1/MPL1.1) on the generation side.*
>
> The main goal is to provide complete and high quality software to manage PDF content. For all the reasons exposed in the GoalsAndMotivations webpage, we really *need* free access to the PDF technology. We plan to appoint the GNU PDF project as a FSF high priority project. We are also working to get funds from governments in order to pay developers. Under of our point of view, it is a quite urgent issue.
>
> As you point out, there is existing software that may provide something like that, but we decided to start a new project from scratch due to some requirements we have in mind:
>
> - completeness
> - portability
> - efficiency
> - robustness regarding legal issues
>
> We did a deep research to the existing free software packages implementing the PDF file format. For one reason or another, we decided to no reuse those programs.
>
> Let me quickly explain what our reasons are.
>
> Ghostscript is a marvelous piece of software. Its coverage of the postscript specification is really good, and its capacity to run even in a toast machine is impressive. But as you know (surely better than any other human being :)) the

ghostscript codebase is also huge and quite complex. Note that I am not saying it is too complex for the tasks it implement: as Peter Deusch says, to use the gs allocators with GC in the C level is not a happy thing, but we dont know a better way to do it. I agree with that. Ghostscript is complex just because it implement complex things. And i consider that complexity level is very well managed in ghostscript. Again, you are one of my hacker-heros :)

But the complexity associated with postscript interpretation is not needed for PDF interpretation. We prefer to work in a lightweight PDF interpreter. As long as i know (i may be wrong) similar reasons led the ghostscript people to launch GhostPDF and the MuPDF+Fitz prototype.

We also had in mind other minor considerations for decide not to use ghostscript for this task. The PDF interpreter distributed with ghostscript is written in postscript. It is not easy to find hackers capable to (or willing to) write postscript. Also, it is difficult for other applications to interact with ghostscript in order to, for example, extract information from a PDF file. The libextractor maintainer decided to use poppler for this reason (and still he is not very happy, since poppler has some difficulties that I will address later). The GNU PDF library should provide GNU (and free software in general) software a convenient access to the Adobe technologies regarding PDF.

We also considered to use xpdf or the poppler library. Almost all free software viewers supporting PDF are using that library, after all. It works and is actively maintained. But we found enough arguments to not use it. First of all, there is the portability issue. poppler is written in C++ and extensively uses the standard template library. If it is difficult to write portable C code, to use C++ is to call for portability problems. Someone may want to embed the gnu library in an embedded device, for example. There is another reason against to use C++ for the library: the vast majority of the GNU system is written in C, and one of the goals of the library is to provide convenient PDF support to other GNU packages.

We are using a bored but we hope effective method to achieve completeness: to design and implement in "width" rather than in "deep". Before to think to implement the lexer or the parser, for example, we want to have support for all the filters (even the rarely used ones, and including the encryption ones), all the structured PDF objects (including the PDF functions, all its types), etc. We dont want to pass to the next chapter of the specification (in a figured way, you know the PDF spec is not exacly linear) until we have complete support of the previous ones. Under this point of view, the objective of the GNU PDF project may differ enough to the objective of ghostscript and poppler to considerate a new implementation. As we see it, the ghostscript goals are more oriented to good postscript support rather than to good PDF support (it is a bigger and difficult objective!). In a similar way, the objective of the poppler project seems to be more visualization-oriented than to provide good interfaces for PDF editing.

Finally, we also directed our attention to MuPDF+Fitz. Again, we detected some degree of divergence in objectives: the author of both mupdf and fitz seems to be more interested in a superb graphics library implementation (Fitz) rather than its interface with PDF (MuPDF). It is a wonderful task and i think he is doing a very good work in adapting Fitz to support several distinct imaging models (such as the Metro support). I would not be able to do such a good work.

These arguments lead José E. Marchesi to create the GNU PDF project, and leaving the maintaining tasks of the GNU gv and GNU ghostscript packages. The first working efforts on designing an appropriate architecture for the GNU PDF Library began. It was decided to take the same approach than the Adobe SDK with respect to layers: the library would consist of the base layer, the object layer, the document layer and the page contents layer. This would made the GNU PDF Library to provide similar services than the Adobe SDK did, and would made easier for final users to modify their Adobe SDK depending applications to use the GNU PDF Library facilities as a replacement.

Despite having a considerable delay of two years, the GNU PDF project keeps working hard to bring a full and conformance-aware support to PDF technologies to the Free Software community.

2.3 Joining GNU PDF

This section describes the process of joining the GNU PDF project. It can be read as a guide for any volunteer interested in collaborating in GNU PDF, as well as a record of the steps performed to do so in the context of this FDP.

Joining the GNU PDF project occurs basically in three mediums: the *information for newcomers* page, the `#pdf` IRC channel at `irc.freenode.net`, and the `pdf-devel` mailing list.

2.3.1 GNU PDF Information for Newcomers

The *GNU PDF Information for Newcomers* roadmap can be found in several places across the Web, most of them under the Free Software Foundation [19]. The official one is located in the GNU PDF website [35].

This page thanks the interest of any user visiting it, and lists ten basic tips with useful information to get started:

1. *Getting a copy of the sources* guides the user to get a local copy of the development branch.

2. *Getting familiar with GNU Bazaar* points the reader to `bzr` tutorials, which is the version control tool used in the project.

3. *Subscribing to the development mailing list* encourages the volunteer to officially present himself or herself to the rest of the GNU PDF community, and to allow the community know how he or she can contribute.

4. *Getting familiar with Savannah* explains the GNU PDF project in this central point of GNU software.

5. *Getting familiar with the library* suggests the user to take a look to the design of the library and the source code.

6. *Getting familiar with the GNU standards* points to official GNU coding standards, which are currently used for writing GNU PDF source code.

7. *Getting familiar with the coding conventions* points to extra coding requirements of GNU PDF.

8. *Taking a task to work on* invites the contributors to choose a first task to develop (see section 2.6).

9. *Signing papers* gives instructions about licensing issues of the collaborations.

10. *Sending patches for inclusion* explains how patches are sent, reviewed and used to update the development branch.

2.3.2 `#pdf` channel at `freenode.net`

Whilst this IRC channel is not listed in the *GNU PDF Information for Newcomers* (see 2.3.1), it is a great place to interact with GNU PDF members.

Actually, users in `#pdf` do not only chat about GNU PDF developments, but also on general programming, creative hacks, Free Software, PDF technologies, and so on. Additionally, many users not directly interested in GNU PDF or free software enter the channel to ask some low-level questions or hacks they need in their PDF files. As many members of the GNU PDF community have a deep knowledge on how PDF works, most of these problems can be solved.

Of course, `#pdf` is the central point for programming discussion regarding GNU PDF tasks. Any contributor can always get advices, reviews or constructive criticism on his or her code.

Usually, and this was the case for this FDP, contributors are first enroled through the `pdf-devel` mailing list, and after that they enter the `#pdf` channel to have their first, more relaxed conversations.

2.3.3 Entering the mailing list

Joining and presenting oneself the `pdf-devel` mailing list is the third step in the *GNU PDF Information for Newcomers*, and also the official mechanism which volunteers use to make the GNU PDF community know how they can contribute to the project.

The list is used for official communications, general discussion and issues about the GNU PDF project itself, but rarely for extensively commenting hacks or programming (the IRC channel is the place for this). The most important use of `pdf-devel` is sending and making publicly available *patches*, and replying to these patches with reviews, comments or acceptance notifications.

There also exists another list, `pdf-tasks`. All contributors should subscribe to it, but never send messages to it, as it is set to automatically send reports from the task management system when, for example, someone picks up a new task or updates its development status (see section 2.5.5).

2.4 Licensing

The GNU PDF project, as the rest of GNU projects under the Free Software Foundation, Inc., requires its contributing members to sign some papers.

2.4.1 Copyright assignment

The copyright assignment is a necessary step for all contributors of a GNU project. Despite the broad right of distribution conveyed by the GPL, enforcement of copyright is generally not possible for distributors: only the copyright holder or someone having assignment of the copyright can enforce the license. If there are multiple authors of a copyrighted work, successful enforcement depends on having the cooperation of all authors.

In order to make sure that all FSF's copyrights can meet the recordkeeping and other requirements of registration, and in order to be able to enforce the GPL most effectively, FSF requires that each author of code incorporated in FSF projects provide a copyright assignment [18, 38].

This assignment is sent by postal mail by the FSF. The envelope and contents of the letter are shown in figures A.1 and A.2, respectivelly, in appendix A. The copyright assignment is a double-sided paper that contains all terms the contributor must agree (see figures A.3 and A.4). Once signed, the letter was sent back to the FSF by postal mail to its headquarters in Boston, MA, USA.

2.4.2 University disclaimer

There is a second type of paper that the FSF requires its project contributors to sign: a disclaimer of any work-for-hire ownership claims by the programmer's employer. This is a guarantee for the FSF that the programmer's employer transfers appropriatelly intellectual rights so he or she can not claim for the contributed code of one of his or her employees. That way FSF can be sure that all the code in FSF projects is free code, whose freedom we can most effectively protect, and therefore on which other developers can completely rely [38].

In this FDP case, the FSF has also an adapted disclaimer for universities. As universities may claim for rights on developments performed by its students or professors, this is a very similar case to the employer / employee described above. Figures A.5 and A.6 in appendix A show the contents of this disclaimer, which was appropriately filled and sent to the FSF.

2.5 Project methodology and organisation

This section consists of a collection of items and subjects that shall be considered by any GNU PDF contributor. It describes some methodological and organisational issues regarding the library architecture, source retrieval and GNU PDF installation on most systems, the development branch directory, the patch and reviewing process and the task management system.

2.5.1 Library architecture

The GNU PDF Library Architecture Guide [25] contains a description of the architecture (both external and internal) of the GNU PDF Library.

The GNU PDF Library contains several layers, shown in figure 2.1.

- **Base Layer**. This layer implements basic functionals such as memory allocation, fixed-point arithmetic, interpolation functions, geometry routines, character encoding and access to the filesystem. The base layer is responsible for providing common system-independent abstractions to other parts of the library.

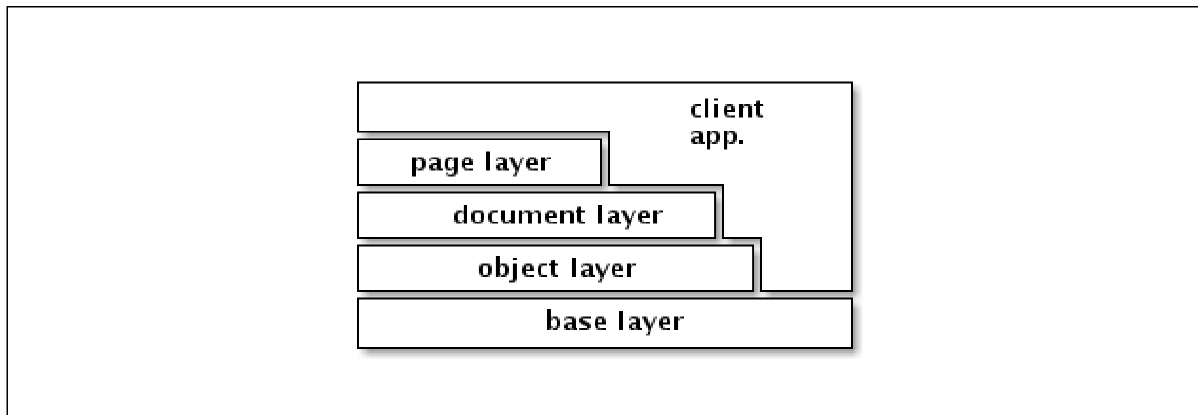


Figure 2.1: Layered architecture of the GNU PDF Library.

- **Object Layer**. This layer implements the concepts of PDF objects and PDF documents as a structured hierarchy of objects. An API is provided to manipulate that structure and the objects that are part of it. Since the object hierarchy can be quite complex a garbage collection mechanism is provided to the client of the layer.
- **Document Layer**. This layer implements the concept of PDF documents as a collection of pages, annotations, fonts, sounds, 3d artwork, discussion threads, forms, etc. It is implemented on top of the object layer. An API is provided to manipulate those abstractions.
- **Page Layer**. This layer implements several abstractions that represent the contents of a page in a PDF document: text, lines, arcs, bitmaps, etc. This layer also provides rasterized bitmaps with page contents, using some graphics library. An API is provided to both read and write page contents.

Some remarks:

- Each layer is built using the API exported by the underlying layers.
- A layer is made up of one or more modules.
- A client application has access to all layers.
- Each layer export an API that gives access to the functionality it implement to clients.
- The terms *procedure* and *function* are used to mean the same thing.

Most developing effort is concentrated in the base layer. The base layer of the GNU PDF Library provides system-independent access to several facilities used by the upper layers of the library. The modules in this layer are quite generic, and are available to the user of the library. Some modules within the base layer make use of the facilities implemented in other modules within the base layer (such as allocation or error functions). A diagram showing its components is shown in figure 2.2.

- **Memory Allocation Module**. This module provides system-independent memory allocation/deallocation.

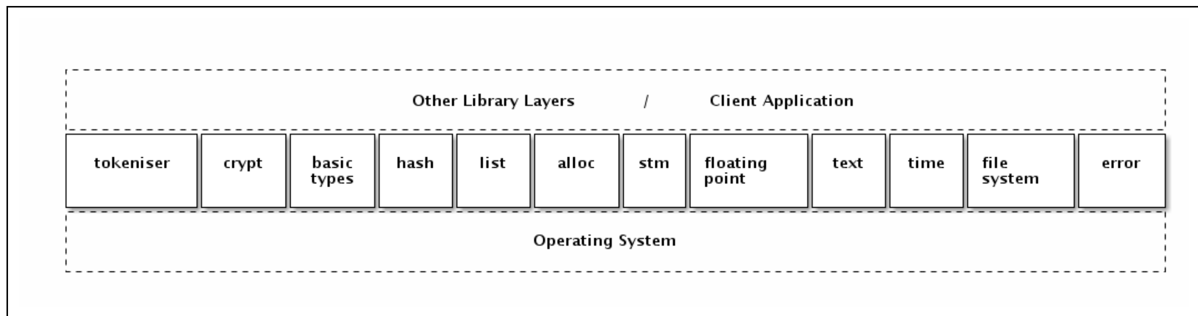


Figure 2.2: Architecture of the Base Layer.

- **Basic Types Module**. This module provides a system-independent implementation of basic data types such as signed and unsigned integers, constants, etc.
- **Hash Module**. This module provides an implementation of associative tables using several hashing algorithms.
- **List Module**. This module provides an implementation of dynamic lists and vectors.
- **Stream Module**. This module provides access to a range of streams (including compressed streams) used within PDF files. The streams are buffered and support filtering for both read and write operations.
- **Floating Point Arithmetic Module**. This module provides system-independent floating point real numbers and several related facilities such as matrix and points manipulations, interpolation routines, real to string and string to real conversion and rounding.
- **Text Module**. This module provides access to a text abstraction as an encoded sequence of characters. Several text encodings are supported and conversion functions to change the encoding of a text are provided.
- **Time Module**. This module provides facilities to manipulate calendar dates, time arithmetic and time spans.
- **Filesystem Module**. This module provides facilities to access filesystem objects (files, directories, file permissions, etc) in a system-independent way.
- **Error Module**. This module provides facilities to manage error types, error signaling, error descriptions, etc.
- **Crypt Module**. This module provides de/encryption and cryptographic hashing facilities.
- **Tokeniser Module**. This module implements a stream tokeniser to read in PDF tokens from a base-level stream and a token writer that can write PDF tokens into a stream.

Some of these modules are going to be used in the implementation of the UUID module, the specification of the conformance module, and the PDF object parser:

- The UUID module may use resources of the Memory Allocation Module and the Basic Types Module, whithin which it is located.
- The conformance module may be located in the base layer, using the Error module.
- The PDF object parser may use resources of the Memory Allocation Module, Basic Types Module, Stream Module, Text Module, Filesystem Module, Error Module and Tokeniser Module.

2.5.2 Sources retrieval and GNU PDF installation

Volunteers are free to download the main branch of the GNU PDF Library source code. To do so, they can follow the little guide included in this section [47]. However, only project maintainers have permissions to apply modifications on the main branch throught the patch mechanism, as explained in section 2.5.4.

All GNU PDF library source is managed with the bazaar version control system, so the first step is to install the `bzr` package. It can be installes from source, from a pre-compiled package, or from a preferred repository. For the latter, and assuming a Debian-style system, it can be installed by just typing on a terminal (with permissions):

```
apt-get install bzr
```

Answering yes to APT will install the packages and all dependencies needed. Now it is time for retrieving the GNU PDF Library sources:

```
bzr branch bzr://bzr.sv.gnu.org/pdf/libgnupdf/trunk
```

Wait a while, and source will be downloaded to ./trunk. Step inside this directory:

```
cd trunk
```

The `autogen.sh` script will prepare the build, but it depends on the `autoconf` and `libtool` packages, so after installing them the library can be bootstrapped:

```
apt-get install autoconf libtool
sh autogen.sh
```

After some messages from the `libtool` library the source is ready to configure, but usually some dependencies are not fulfilled at this point: `zlib`, `libgpg-error`, `libgcrypt`, `uuid-dev` and `libcheck`. Except `libcheck`, the rest of the required libraries are available in the Debian repos (and usually in the rest of distributions):

```
apt-get install zlib1g-dev libgpg-error-dev libgcrypt11-dev uuid-dev
```

The GNU PDF library requires the SVN source of `libcheck`, a C unit test framework [7], to ensure the latest version of this library. Obviously the `subversion` package is required, and then sources retrieval, configure, compile and install can be performed (as root):

```
apt-get install subversion
cd ~
svn co https://check.svn.sourceforge.net/svnroot/check check
cd check/trunk/
autoreconf -i
./configure
make
make install
```

At this point, all GNU PDF library requirements are met:

```
cd ~/trunk/
./configure
make
make install
```

This will install the GNU PDF library in the default location. In most cases, compiled library objects can be found issuing:

```
ls /usr/local/lib
```

2.5.3 The development branch

The root directory of the development branch is depicted in figure 2.3.

- The root `trunk/` directory contains all the library resources. Important files here are the GNU Build System files (`autogen.sh`, which calls `autoreconf`, `configure.ac`, and so on), the `ChangeLog` and some `README` files.
- The `build-aux/` subdirectory contains several programs and utilities used to build the library. These programs are not intended to be installed in the target machine. These programs include a C source generator for the Unicode Character Database; files from the `pmccabe2html` *gnulib* module providing the generation of cyclomatic complexity reports in both mediawiki markup and html (see section 2.5.6); or a documentation generation script.
- The `doc/` subdirectory contains all the GNU PDF Library documentation, mainly in the *texinfo* [37] format. The most important documents are the *GNU PDF Library Reference Manual* [28], the *GNU PDF Library Hackers Guide* [27], and the *GNU PDF Library Architecture Guide* [25].
- The `lib/` subdirectory contains some low-level static libraries and headers.
- The `m4/` subdirectory contains `m4` macros used by `autoconf`.
- The `prmgt/` subdirectory contains some useful scripts in Python and Emacs Lisp for several purposes (for example, `authors.el` generates automatically the `AUTHORS` file of the root directory).

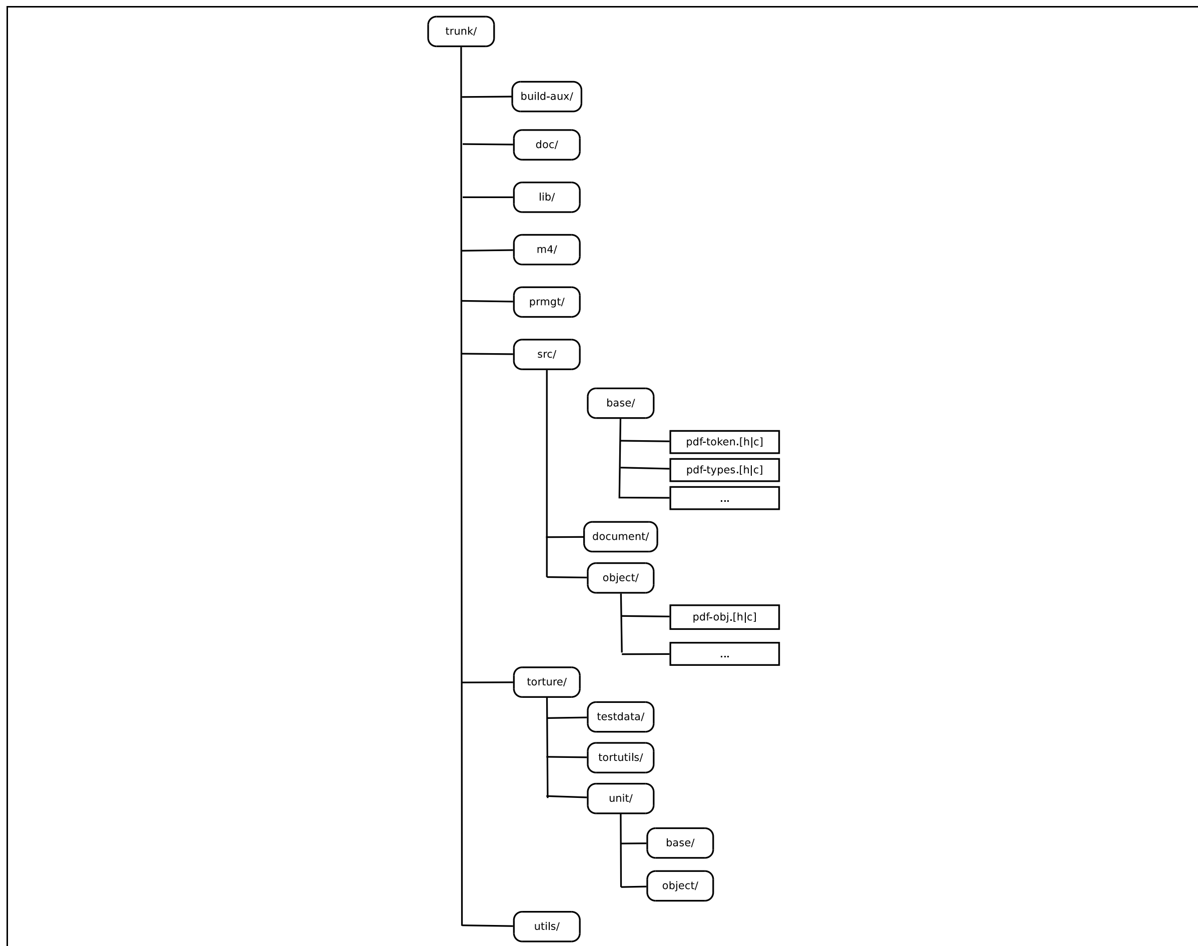


Figure 2.3: Source tree directory of the GNU PDF Library.

- The `src/` subdirectory contains the source code of the GNU PDF Library. Some files included are needed by `autoconf`, but the source code itself is distributed in a subdirectory hierarchy. Each subdirectory corresponds to a layer of the library. Inside each layer subdirectory, code and header files are organised with the syntax `layer/pdf-module-submodule.[h|c]` (for example, `base/pdf-types-uuid.h` refers to the UUID submodule, in the types module, in the base layer).

- The `torture/` subdirectory is also known as the *torture chamber*. It contains the test framework of the library. The architecture of the test framework, which for the GNU PDF project is *libcheck*, can be found in chapter 3, section 3.6.1.

- The `utils/` subdirectory contains some tools which make direct use of some part of the library. For instance, `pdf-filter.c` exploits the filtered streams, or `pdf-tokeniser.c` uses the tokeniser module to extract tokens from an input PDF file.

2.5.4 Iterations, patch proposals and review process

The GNU PDF project has an iterative, distributed and reviewed source updating system. Iterative means that the whole library code is modified over and over again, improving its

architecture, specification, design, implementation and documentation. Distributed means that, though having a central repository for version control, the source code is distributed across the network in all contributors' hosts, who send improvements to the main branch. Reviewed means that not all sent improvements are automatically applied; changes with respect to the current branch are evaluated and discussed by the whole GNU PDF community in the `pdf-devel` mailing list. The result of this discussion is either an approval (and then the patch is applied), or a reject (with suggestions on how to correct or improve the code). Only admins have permissions to write to the development branch.

This system has the advantage of improving the overall quality of source code, as it empowers the Linus' Law [49]. However, in a not very large community, which is the case for the GNU PDF project, these mechanics slow the development cycle.

2.5.5 Tasks management

The GNU PDF project uses a task pool to manage the tasks of the development group. Extensive documentation about the task pool dynamics can be found in the GNU PDF Reference Manual [32]. In brief, the tasks in the pool follow the worflow depicted in figure 2.4.

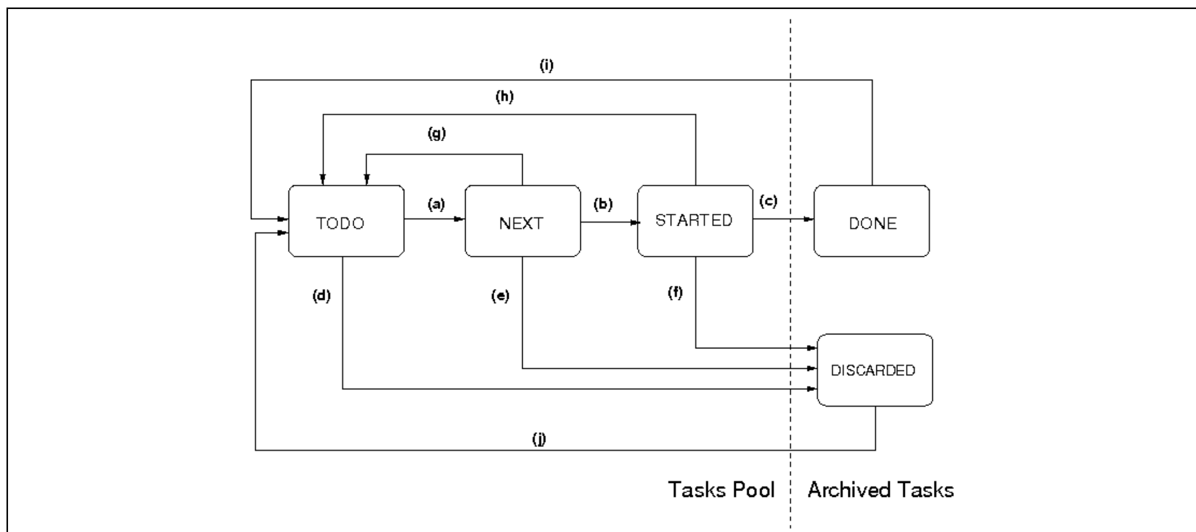


Figure 2.4: Tasks workflow.

- **TODO**. The task is into the tasks pool but it not ready to be performed.
- **NEXT**. The task is into the tasks pool and it is ready to be performed by a developer.
- **STARTED**. The task has been started by a developer but it is not finished.
- **DONE**. The task is archived and succesfully performed.
- **DISCARDED**. The task is archived but it has been discarded.

Note that a task may also be assigned to someone or unassigned:

- **TODO and unassigned**. Dependency issues, no one in charge.

- **TODO and assigned**. Dependency issues and someone showed interest.
- **NEXT and unassigned**. Can be started, nobody interested.
- **NEXT and assigned**. Someone is interested but can't work on it for personal issues.
- **STARTED and assigned**. Someone is working on the task (no matter the time it takes).

The tasks pool is implemented with the *flyspray* task manager (see figure 2.5). It is installed in `http://gnupdf.org/flyspray`. As described below in section 2.6, the UUID (*Universal Unique Identifiers*) module was the first task chosen to work on. To do so, status of the task *Development of the UUID module* was changed from `NEXT` to `STARTED` in the *flyspray* of the project, noticing that someone had began to work on it.

The rest of documentation regarding tasks management can be found in the GNU PDF Reference Manual [32].

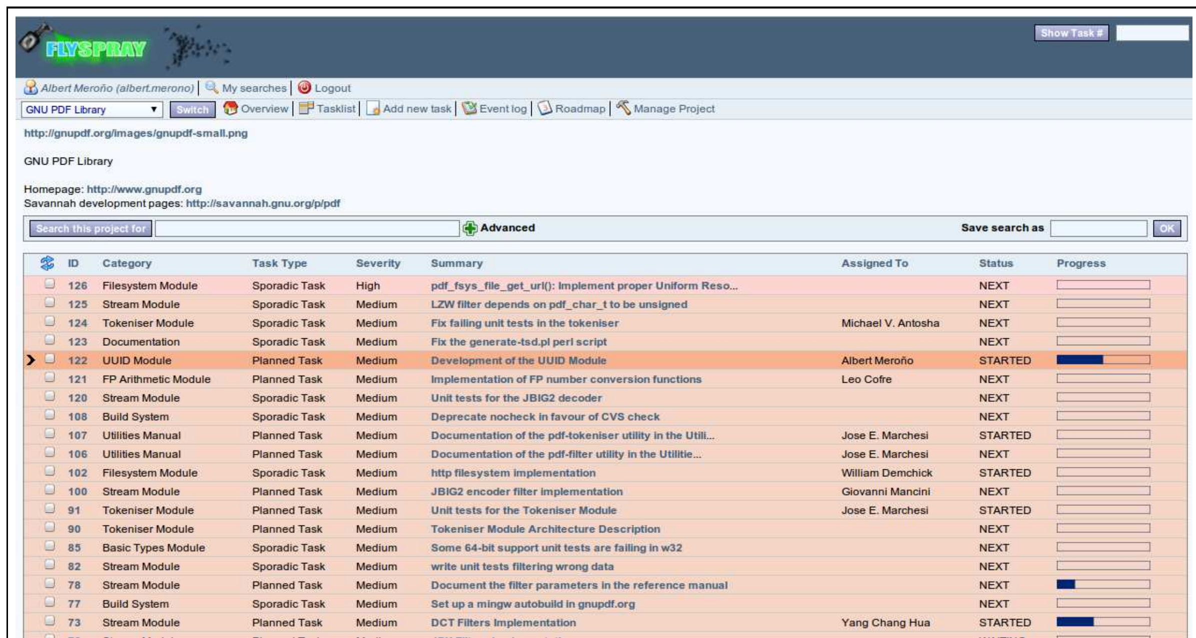
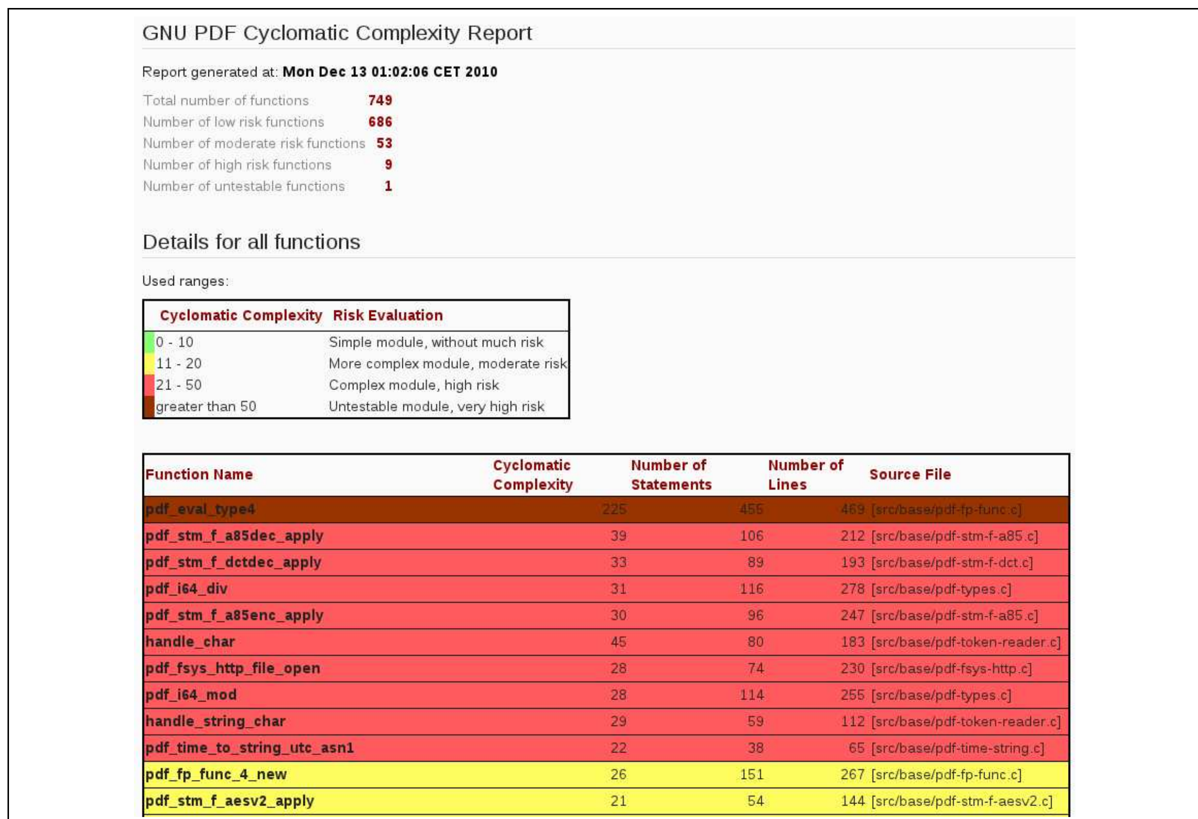


Figure 2.5: The *flyspray* task management system. Task #122 was changed to reflect that someone had started its development.

2.5.6 Library quality

The GNU PDF Library has some tools to automatically evaluate the overall quality of the library. These quality tests are encoded in scripts in the source directory, and are run periodically in the main project server. The most important reports are the *Cyclomatic complexity report*, the *API Consistency report*, and the *Unit Testing Report*.

The Cyclomatic Complexity Report [26] (see figure 2.6) contains a list of all the functions implemented so far in the library and its McCabe number [58], a direct metric that tells about the cyclomatic complexity of the code module (*i.e.* the number of execution paths).



GNU PDF Cyclomatic Complexity Report

Report generated at: **Mon Dec 13 01:02:06 CET 2010**

Total number of functions	**749**
Number of low risk functions	**686**
Number of moderate risk functions	**53**
Number of high risk functions	**9**
Number of untestable functions	**1**

Details for all functions

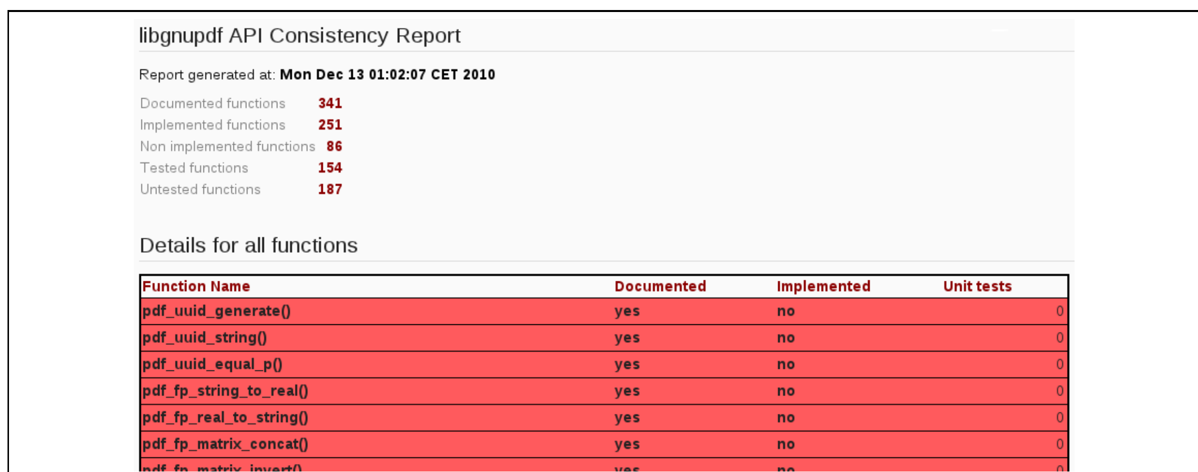
Used ranges:

Cyclomatic Complexity	Risk Evaluation
0 - 10	Simple module, without much risk
11 - 20	More complex module, moderate risk
21 - 50	Complex module, high risk
greater than 50	Untestable module, very high risk

Function Name	Cyclomatic Complexity	Number of Statements	Number of Lines	Source File
pdf_eval_type4	225	455	469	[src/base/pdf-fp-func.c]
pdf_stm_f_a85dec_apply	39	106	212	[src/base/pdf-stm-f-a85.c]
pdf_stm_f_dctdec_apply	33	89	193	[src/base/pdf-stm-f-dct.c]
pdf_i64_div	31	116	278	[src/base/pdf-types.c]
pdf_stm_f_a85enc_apply	30	96	247	[src/base/pdf-stm-f-a85.c]
handle_char	45	80	183	[src/base/pdf-token-reader.c]
pdf_fsys_http_file_open	28	74	230	[src/base/pdf-fsys-http.c]
pdf_i64_mod	28	114	255	[src/base/pdf-types.c]
handle_string_char	29	59	112	[src/base/pdf-token-reader.c]
pdf_time_to_string_utc_asn1	22	38	65	[src/base/pdf-time-string.c]
pdf_fp_func_4_new	26	151	267	[src/base/pdf-fp-func.c]
pdf_stm_f_aesv2_apply	21	54	144	[src/base/pdf-stm-f-aesv2.c]

Figure 2.6: Cyclomatic Complexity Report.

The API Consistency Report [24] (see figure 2.7) contains the relation of documented (specified) functions and the relation of implemented functions. Ideally all documented functions should be implemented. Hence, it shows the gap between the library specification and the library implementation. Additionally, the number of implemented tests for each function is also shown.



libgnupdf API Consistency Report

Report generated at: **Mon Dec 13 01:02:07 CET 2010**

Documented functions	**341**
Implemented functions	**251**
Non implemented functions	**86**
Tested functions	**154**
Untested functions	**187**

Details for all functions

Function Name	Documented	Implemented	Unit tests
pdf_uuid_generate()	yes	no	0
pdf_uuid_string()	yes	no	0
pdf_uuid_equal_p()	yes	no	0
pdf_fp_string_to_real()	yes	no	0
pdf_fp_real_to_string()	yes	no	0
pdf_fp_matrix_concat()	yes	no	0

Figure 2.7: API Consistency Report.

The Unit Testing Report [29] (see figure 2.8) contains the results of automatically running

the unit tests for the library.

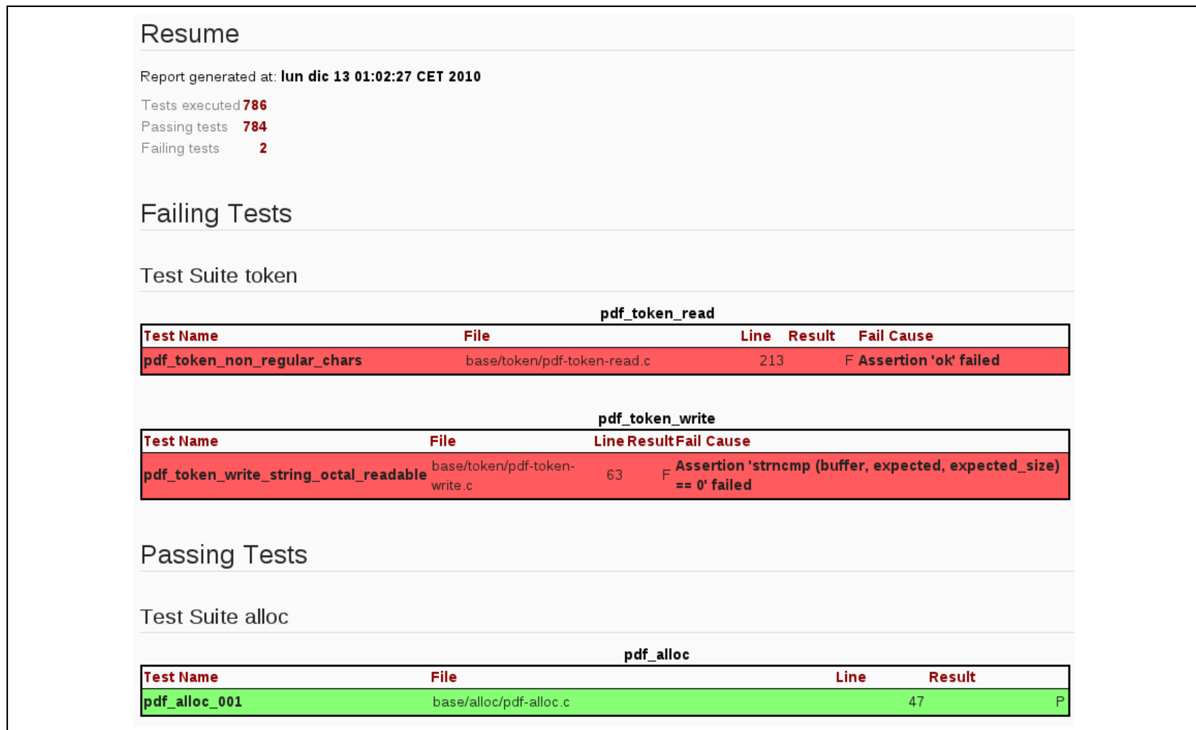


Figure 2.8: Unit Testing Report.

2.5.7 Tools

This section briefly describes the most important tools for any programmer contributing with the GNU PDF Library.

GNU Bazaar [41], also known simply as `bzr`, is the version control system of the project. All source branches, and all common associated operatins like commiting, updating, upgrading or even generating patches, are managed with it. Newcomers are strongly encouraged to get familiar with this tool before contributing.

The *GNU Build System* [20, 61], also known as `autotools` is a suite of programming tools designed to assist in making source-code packages portable to many Unix-like systems. Making a software program portable can be difficult. The C compiler could be different. Some common C functions could be missing, have another name, be declared in a different header and so on. This can be handled by enabling different pieces of code in sources using preprocessor directives such as `if`, `ifdef` and others. But the user would then have to define all this tuning for himself which is not easy as there are lots of systems with lots of variations. The `autotools` are designed to solve this. Additional details can be found in chapter 3, section 3.3.3.

The *Emacs editor* is not a must for GNU PDF hacking, but as happens with most of the GNU system, many development tools and aids are highly focused on it. Morever, GNU and GNU PDF coding standards are implemented almost by default in Emacs. GNU PDF has lots of scripts written in Emacs Lisp that help the programmer in many daily tasks.

2.6 The first task choice

On one hand, and as stated in the GNU PDF Library Hackers Guide [27], newcomers should take a first `NEXT` task (see 2.5.5) to work on. On the other hand, this FDP collaboration also needed a first task to contribute. However, the features of this first task were not arbitrary. Moreover, it can be stated that the first task of the FDP had the following additional requirements:

1. The primary goal was to learn how to collaborate and work with and for the GNU PDF community. This corresponds with the first general goal of the FDP.
2. The first task should not be extensive in time, as other FDP major developments should be performed after it, like conformance related tasks and a parser for PDF.
3. As a consequence of the two preceding requirements, the provided source code for the first task should not be extensive in terms of lines of code or functionalities. The priority was to learn the development methodology, the reviewing process, and the project's development tools.
4. The chosen task should include more stages other than implementation. Requirements analysis, specification and design decisions were desirable facets to be encountered while developing the first task, apart from implementation.
5. The task should allow to develop some testing stage of the developments performed, in order to get familiar with the library testing framework.

After studying all options offered by the *flyspray*, it was decided to choose the task entitled *Development of the UUID module* (see figure 2.5), as it fit well with all requirements. The development of the UUID module is detailed in chapter 3.

Chapter 3

The UUID module

This chapter describes the development process of the UUID (*Universally unique identifiers*) module. Reading of this chapter may be confusing if the reader does not keep in mind what the main goals of the UUID module development are: to learn the development methodology of GNU PDF, to assimilate the use of important tools of the project, such as the GNU Build System, and to contribute with some source code. For these reasons, after a brief introduction the chapter consists of detailing the iterations performed, rather than going through the classical waterfall model stages. One of the goals of this chapter is to suggest to the reader the incremental cycle implicitly used in GNU PDF. This cycle is later presented in chapter 8.

3.1 Universally unique identifiers

A **universally unique identifier** (**UUID**) is an identifier standard used in software construction, standardized by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE) [70].

The aim of UUIDs is to enable distributed systems to uniquely identify information, without requiring any central coordination. Thus, anyone can create an UUID and use it to identify something (*e.g.* an instanciated struct or a row in a database) with a high confidence that the same UUID will not be used by anyone for anything else again. Therefore, information tagged with UUIDs can be merged in a single database without requiring to solve name conflicts. A distributed application may generate data and store it in several locations; when data needs to be merged at some intervals of time, UUID can then be used as the primary key.

UUIDs are required in the GNU PDF Library to generate IDs for the PDF documents. Additionaly, UUIDs shall be used in any occasion a module requires to uniquely identify information.

In the context of the GNU PDF Library UUIDs are seen as basic data types. Hence, the UUID module is in fact a submodule, a piece of the basic types module, which is located in the base layer. This is shown in figure 3.1.

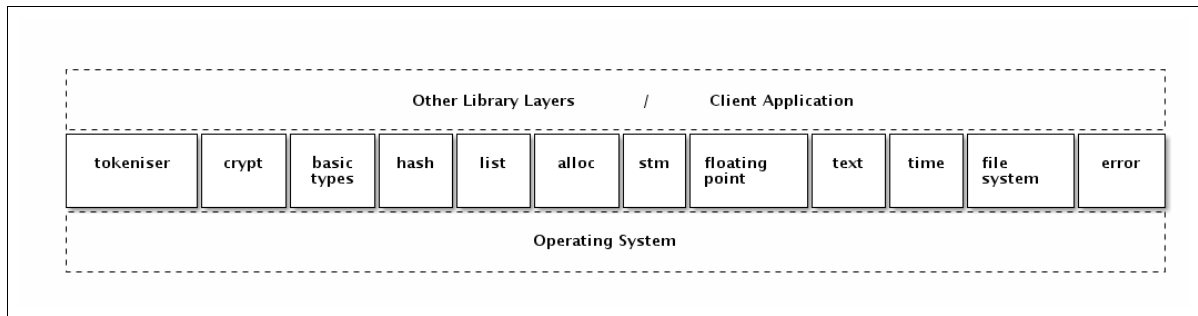


Figure 3.1: A diagram of the base layer architecture. The base layer operates on top of the operating system, and receives calls coming from other layers above it, and from client applications. The UUID submodule is located in the basic types module.

3.1.1 Internal structure

Each UUID is a hexadecimal-coded ASCII sequence composed by the following fields, separated by the ASCII hyphen-minus, 45 character, except between the VariantAndClockSeqHigh and ClockSeqLow (see table 3.1[15])[28].

Field name	Bytes	Hex digits	Description
TimeLow	4	8	The low field of the timestamp
TimeMid	2	4	The middle field of the timestamp
VersionAndTimeHigh	2	4	The high field of the timestamp multiplexed with the version number
VariantAndClockSeqHigh	1	2	The high field of the clock sequence multiplexed with the variant
ClockSeqLow	1	2	The low field of the clock sequence
Node	6	12	The spatially unique node identifier

Table 3.1: The fields of a UUID, with corresponding sizes in octets and hexadecimal digits, and a description of each.

To generate these fields, the following data must be retrieved from the host generating the UUID:

- **Timestamp**. The timestamp is a 60-bit value, which is represented by Coordinated Universal Time (UTC) as a count of 100 nanosecond intervals since 00:00:00.00, 15 October 1582 (the date of Gregorian reform to the Christian calendar).
- **Clock sequence**. The clock sequence is used to help avoid duplicates that could arise when the clock is set backwards in time or if the node ID changes.

- **Node**. The node field consists of an IEEE 802 MAC address, usually the host address. For systems with multiple IEEE 802 addresses, any available one can be used.

Additional details can be found at RFC 4122 [15].

3.2 Iterative development

GNU PDF uses an iterative development methodology, as described previously in section 2.5. Putting this into practice, the development process of this module was structured in a number of short, quick and incremental iterations. Each iteration consists of phases. Phases involved in a given iteration depend on what exactly needs to be improved with respect to previous milestones.

Considering the review process involved it is difficult to previously plan a fixed number of iterations. The eyes of the community (more exactly, the eyes of those subscribed in the mailing list) recieve proposed changes from volunteers in form of patches, and freely critizise them. Usually the same volunteer that sent a reviewed patch fixes it with suggestions of the community, but this does not exclude more reviews. Moreover, the implementation of new features or simply correcting mistakes can lead to new improvement suggestions, in form of bad memory usage, unefficient code, unreadable code, non portable code, non thread safe code, and so on. In the long term, progressive patches for a given task require less work to do, which means that code improves its quality in the review process. This cycle ends when no more suggestions are proposed, and the patch is applied to the official development branch. Code applied this way can not be considered as final code in any way. Further reviews may occur (*e.g.* someone writes a unit test that makes bugs arise; a module using submitted code fails to work correctly because of an error in that code), and improvement suggestions can be made at any further time.

In the UUID module case, four iterations were needed to get two approvals from the GNU PDF community. The following subsections describe the work done in each one of these iterations.

3.3 First iteration

The first iteration had the goal of producing a first patch proposal.

3.3.1 Requirements analysis

In this subsection the analysis of requirements for the development of the UUID module is presented. First, the sources for requirements, which can be defined as places were stakeholders wrote critical requirements for this module, is described. Finally, as a result after the study of these requirements sources, the complete list of requirements is presented, distinguishing between functional and non-functional requirements.

Requirements sources

This section describes the sources for requirements regarding the UUID module. The very first requirements for almost any task are located in the task description, which can be accessed

by developers just clicking on that task in the *Flyspray* manager. The *Flyspray* manager was previously described in section 2.5.5.

In the task description view several fields can be modified in order to reflect the current condition of the task (*e.g.* changing its status or writing some comments). One particular field gives some *details*: guidelines that the originator of the task wrote to help the task taker to start with his or her work.

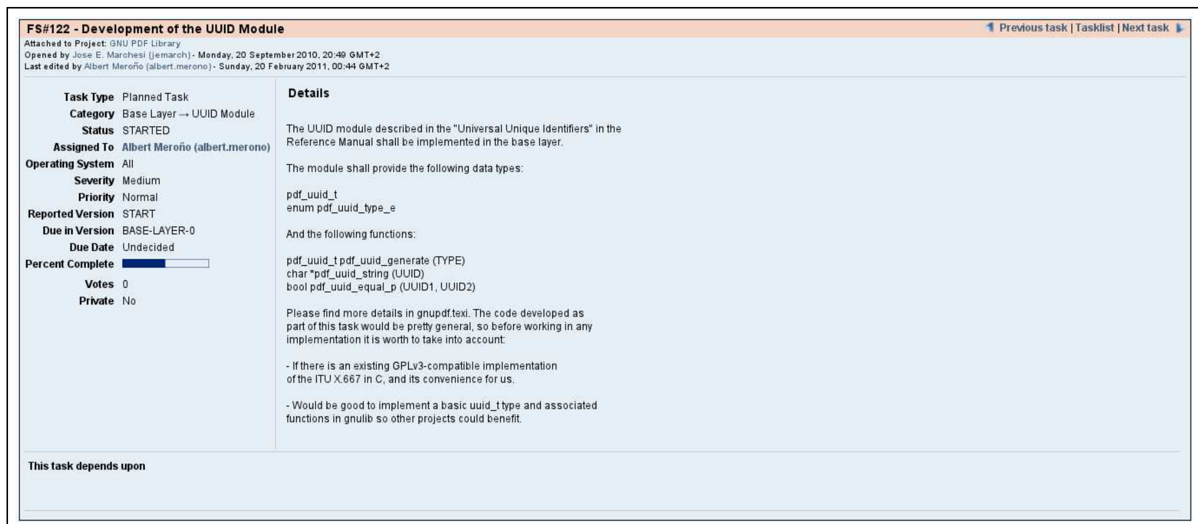


Figure 3.2: The *Flyspray* task management system. The details view is the starting point for studying task requirements.

In the case of the UUID module development, these guidelines provided the contents shown in figure 3.3.

As many other tasks in the *Flyspray*, these details usually point the reader to the GNU PDF Reference Manual to get more information. This sets up the Reference Manual as the second requirements source for analysis. So, apart from the *Flyspray* manager, the GNU PDF Library Reference Manual [28] was studied for data types and functions that this module should provide through the library API. Data types are shown in figure 3.4, and functions are described in figures 3.5, 3.5 and 3.5.

Functional requirements

After studying requirements sources, the following functional requirements were found to be needed in the UUID module.

- A **UUID data type**, holding the UUID data. This data type shall conform with the directives of the GNU PDF Library Hackers Guide [27] regarding *abstract data types* (ADT).

- An enumeration of **UUID types**. This enumeration shall list all kinds of UUIDs that the library can generate.

- A **UUID generation function**. This function shall generate an UUID of the specified type.

The UUID module described in the "Universal Unique Identifiers" in the Reference Manual shall be implemented in the base layer.
The module shall provide the following data types:

```
pdf_uuid_t
enum pdf_uuid_type_e
```

And the following functions:

```
pdf_uuid_t pdf_uuid_generate (TYPE)
char *pdf_uuid_string (UUID)
bool pdf_uuid_equal_p (UUID1, UUID2)
```

Please find more details in gnupdf.texi. The code developed as part of this task would be pretty general, so before working in any implementation it is worth to take into account:

- If there is an existing GPLv3-compatible implementation of the ITU X.667 in C, and its convenience for us.
- Would be good to implement a basic uuid_t type and associated functions in gnulib so other projects could benefit.

Figure 3.3: Contents of the *details* field in task #122, UUID Module.

Data Type: `pdf_uuid_t`
Opaque type representing an Universally Unique Identifier.
Data Type: `enum pdf_uuid_type_e`
Type of an UUID:
`PDF_UUID_TIME`
Time-based UUID as defined by ITU X.667.
`PDF_UUID_RANDOM`
Random-based UUID as defined by ITU X.667.
`PDF_UUID_NAME`
Name-based UUID as defined by ITU X.667.

Figure 3.4: Required data types for the UUID Module.

- An **ASCII representation function** of a given UUID. This function shall generate a string with an ASCII representation of the specified UUID.
- A **UUIDs comparison function**. This function shall compare two given UUIDs and return if they are equal or not.

Non-funtional requirements

After studying requirements sources, the following non-functional requirements were found to be needed in the UUID module.

- To be written in C

Function: `pdf_uuid_t`
`pdf_uuid_generate (enum pdf_uuid_type_e type)`
Generate a new UUID of the specified type.
Parameters
type
The type of UUID to generate.
Returns
The generated UUID.
Usage example

```
pdf_uuid_t uuid;

/* Generate a name-based UUID.  */
uuid = uuid_generate (PDF_UUID_NAME);
```

Figure 3.5: First required function for the UUID Module.

Function: `const char * pdf_uuid_string (pdf_uuid_t uuid)`
Return an ASCII string with the printed representation of uuid.
Parameters
uuid
A previously generated UUID.
Returns
A null-terminated buffer containing the printed representation of uuid.
Usage example

```
pdf_uuid_t uuid;

uuid = pdf_uuid_generate (PDF_UUID_TIME);
printf ("The generated UUID: %s\n",
        pdf_uuid_string (uuid));
```

Figure 3.5: Second required function for the UUID Module.

Function: `pdf_bool_t pdf_uuid_equal_p (pdf_uuid_t uuid1, pdf_uuid_t uuid2)`

Determine if two given UUIDs are equal.

Parameters

uuid1

The first UUID to compare.

uuid2

The second UUID to compare.

Returns

A PDF boolean indicating whether both UUIDs are equal.

Usage example

```
pdf_uuid_t uuid1;
pdf_uuid_t uuid2;

uuid1 = pdf_uuid_generate (PDF_TIME);
uuid2 = pdf_uuid_generate (PDF_TIME);

if (pdf_uuid_equal_p (uuid1, uuid2))
{
   /* Extremely unlikely!  */
}
```

Figure 3.5: Third required function for the UUID Module.

- To be ITU.667 compliant
- To be GPLv3 compatible
- If possible, to reuse an existing implementation meeting these requirements

3.3.2 Specification and Design

This subsection shows how the process of transforming requirements into functionalities and behaviours was performed. Three separate parts are included: first, the search for an avaiable, suitable and existing implementation for the UUID module; second, the API offered by the module as a representation for functional requirements; and third, some design decisions that were took after considering the two previous points.

Existing, suitable UUID implementations

The key non-functional requirement of this module was to find an existing implementation of UUIDs suitable for the GNU PDF Library. Such an implementation could be used in the GNU PDF Library's UUID module, instead of one coded from scratch. This section explains how external UUID libraries were found, analyzed and chosen, according with GNU PDF Library requirements.

The flyspray task stated that these hypothetical implementations shall be

- written in C,
- ITU.667 compliant,
- and GPLv3 compatible.

There exist lots of UUID implementations for almost any programming language [70]. However, only the following two implementations written in pure C were found:

- *libuuid*, part of the **e2fsprogs** package.
- *OSSP uuid*, part of the **OSSP** project.

Source code of both software packages was retrieved (see figure 3.6), and a first study of these sources performed.

```
apt-get source libossp-uuid16 libuuid1
```

Figure 3.6: Retrieval of source code for candidate UUID libraries. Commands issued in a Debian GNU/Linux system, using the APT package and source manager to query Debian software repositories.

This confirmed that the implementation language was C. Also, the licensing compatibility was an issue: copyright and general licensing information included in sources file headers for *libuuid* and *OSSP uuid* is shown in appendix B.

As neither these notes nor the project homepages [53, 42] provided reliable proof about the licensing status of these packages, authors were contacted via e-mail. A copy of the e-mail sent

to authors or maintainers of *libuuid*, Theodore Ts'o, and *OSSP uuid*, Ralf S. Engelschall, can be read in figure 3.7. Theodore Ts'o is a very well known Linux kernel hacker, particularly for his contributions to file systems, and primary developer and maintainer of **e2fsprogs**, which includes **libuuid**. UUIDs were first developed in the context of **e2fsprogs**, a user space set of utilities for maintaining ext2, ext3 and ext4 filesystems, and are used there to uniquely identify filesystems and disk drives.

```
Dear Theodore,

I'm a free software developer and currently contributor to the
GNU PDFproject. The library needs an implementation of UUID,
and before coding the correspoing types and funcionalities
we're looking for previous implementations meeting these three
requirements:

- To be written in C
- To be ITU X.667 compliant
- To be GPLv3 compliant

AFAIK libuuid meets the first, but does it meet the last two?

Thank you very much in advance,

Albert Meroño
```

Figure 3.7: E-mail sent to Theodore Ts'o, asking for requirements suitability of the UUID implementation deployed with **e2fsprogs**. An analogous mail was sent to Ralf S. Engelschall for the same reasons with respect to **OSSP uuid**.

As some days went by, only the e-mail sent to Theodore Ts'o got a reply, which is included in figure 3.8.

The license compatibility issue with respect to *libuuid* was then solved: the library is licensed under a BSD license (though originally it was under LGPLv2) compatible with GPLv3 developments, which is the case for the GNU PDF Library.

Ted's mail also helped in the second requirement in the list: ITU X.667 compliance. The ITU X.667 [55] norm was also studied and therefore compared with OSF/DCE specification [16] and RFC 4122 [15]. This comparison was to ensure that *libuuid* compliance with OSF/DCE was extensible to RFC 4122 and ITU X.667, which was the initial interest of GNU PDF.

From an implementation point of view, *OSSP uuid* presented some disadvantages against *libuuid*. The main problem with OSSP code is that it abuses object orientation, though being written in C. A UUID, as shown in section 3.1.1, is just a 128-number that can be easily reprented through an `unsigned char[16]` array. UUIDs are commonly attached to other structures (*e.g.*, **e2fsprogs** uses them to identify file systems), and it is desirable to inline the UUID in the structure itself. *OSP uuid* returns a `uuid *` pointer after a new UUID has been created, pointing to a dynamically allocated object, which holds much more information than the ASCII characters conforming the UUID. If one requirement for the system using these

```
My apologies for the delay in getting back to you.  When I get
busy I don't always read e-mail sent to the thunk.org e-mail
address regularly.

The uuid library is written to follow the OSF/DCE
specification, which was than used as the source material for
RFC 4122 and X.667.  The two specifications are aligned, but
why anyone would pay $$$ for a fourth generation copy of X.667
from some national standards body, when you can get RFC 4122
for free download on the web, is a mystery to me.

The UUID library was originally released under the LGPLv2
license. It has now been released under a BSD license, to
allow Apple to use the library in Mac OS X.  Hence the library
is compatible with GPLv3 code.

I would appreicate changes/improvements to the UUID library be
donated back using the currently-used BSD license, instead of
forked to be in the GPLv3-only universe, although the BSD
license certainly legally allows you to do this.  I don't
consider it a neighborly thing to do, but it's certainly
legally permissible.

Best regards,

                                              - Ted
```

Figure 3.8: E-mail received from Theodore Ts'o, in reply for the requirements questions pointed above.

UUIDs is efficiency or scalability, working with thousands of those UUIDs makes the program run slower, uses more memory and causes memory fragmentation. Finally, generation of new UUIDs in *OSP uuid* involves 4 calls (`uuid_create`, `uuid_make`, `uuid_export` and `destroy`), plus those necessary for the programmer to work with the UUID value. On the other hand, in *libuuid* only one single call is needed to generate the UUID (`uuid_generate`) and one more to get the ASCII representation (`uuid_unparse`).

`pdf-devel`, the mailing list of the GNU PDF Library development group, was reported about this. José E. Marchesi, main developer and co-maintaner of the GNU PDF Library, pointed that Ts'o's library was enough for the project's needs (see figure 3.9). Thus, both licensing and standards compliance of **e2fsprogs** *libuuid* were found fully compatible with GNU PDF Library requirements.

All arguments exposed in this section led to choose *libuuid* as the more appropriate and most suitable UUID library to link with the GNU PDF Library.

```
Hi Albert.

   As for the standard compliance, he pointed that the uuid
   library is written to follow the OSF/DCE specification,
   which was used as source material for RFC 4122 and X.667.
   AFAHK, 4122 and X.667 are aligned. I hadn't got enough
   time to check if the alignment is partial or total
   between both specifications. I think it should be
   performed before deciding if libuuid's code is going to
   be the official uuid module implementation or not.

Tso's uuid library provides both UUID generation and parsing
routines. More than enough for our needs.

   Regarding the license, libuuid was originally released
   under LGPLv2, but currently it is under a BSD license to
   allow Apple to use the library in MacOS X. So the library
   is compatible with GPLv3. Ted would appreciate changes or
   improvements made to libuuid to be donated back using the
   currently BSD license (is that possible after adopting it
   in a GPLv3 project?) instead of forking it in the
   GPLv3-only universe. The BSD license legally allows us to
   do that, but he doesn't consider it "a neighborly thing
   to do".

I don't think that will be a problem.  The simplest way to
go is to write a thin layer in pdf-uuid.[ch] and link with
libuuid.  In case we need to extend the library (quite
unlikely) we can contribute the changes under the BSD
license.
```

Figure 3.9: E-mail reply from Jose E. Marchesi's, pointing that the libuuid library met the requirements needed for the GNU PDF Library. Neither licensing nor standards compliance were considered to be noticeable problems.

Design decisions

After deciding that *libuuid* whould implement the UUID module, some design decisions had to be made. In particular, a study of *libuuid* had to be done before coding anything, in order to decide how to map the existing UUID module API (see figure 3.10) with the *libuuid* API (see figure 3.11).

As shown in figure 3.10, some extra functionalities supported in *libuuid* are not necessary, at least in a first approach, in the implementation of the UUID module of the GNU PDF Library. Moreover, the necessary functions to map into the existing UUID module API are the following:

- `int uuid_compare(const uuid_t uu1, const uuid_t uu2)` in order to implement

```
typedef unsigned char uuid_t[16];

...

/* clear.c */
void uuid_clear(uuid_t uu);

/* compare.c */
int uuid_compare(const uuid_t uu1, const uuid_t uu2);

/* copy.c */
void uuid_copy(uuid_t dst, const uuid_t src);

/* gen_uuid.c */
void uuid_generate(uuid_t out);
void uuid_generate_random(uuid_t out);
void uuid_generate_time(uuid_t out);

/* isnull.c */
int uuid_is_null(const uuid_t uu);

/* parse.c */
int uuid_parse(const char *in, uuid_t uu);

/* unparse.c */
void uuid_unparse(const uuid_t uu, char *out);
void uuid_unparse_lower(const uuid_t uu, char *out);
void uuid_unparse_upper(const uuid_t uu, char *out);

/* uuid_time.c */
time_t uuid_time(const uuid_t uu, struct timeval *ret_tv);
int uuid_type(const uuid_t uu);
int uuid_variant(const uuid_t uu);
```

Figure 3.10: Extract from util-linux-2.17.2/shlibs/uuid/src/uuid.h, which contains the *libuuid* API. The comments point to the source files implementing the listed functions.

the comparison between two given UUIDs.

- `void uuid_generate(uuid_t out)`, `void uuid_generate_random(uuid_t out)` and `void uuid_generate_time(uuid_t out)` in order to implement the generation of name-based and random-based UUIDs.
- `void uuid_unparse(const uuid_t uu, char *out)` in order to implement the function which returns the ASCII representation of a given UUID.

Hence, the mapping between *libuuid* API and the UUID module API was clear at this point. This mapping is shown in table 3.2.

```
typedef struct pdf_uuid_s *pdf_uuid_t;

/* UUID creation */
pdf_uuid_t pdf_uuid_generate (enum pdf_uuid_type_e type);

/* Printed ASCII representation of an UUID */
const char * pdf_uuid_string (pdf_uuid_t uuid);

/* Determine if two UUIDs are equal */
pdf_bool_t pdf_uuid_equal_p (pdf_uuid_t uuid1,
                             pdf_uuid_t uuid2);
```

Figure 3.11: First approach to an UUID module API. Note the ADT encapsulating the UUID, and the three functions providing the required functionalities.

UUID module resource	***libuuid* resource**	**Type**	**Description**
`pdf_uuid_t`	`uuid_t`	Data type	The ADT provided shall contain, at least, all data used by *libuuid* to represent UUIDs.

`pdf_uuid_generate`	`uuid_generate`
	`uuid_generate_random`
	`uuid_generate_time`
		Function	The function that generates UUID uses the generation methods of *libuuid*.

`pdf_uuid_string`	`uuid_unparse`	Function	The function that provides the ASCII representation of an UUID uses the unparse method of *libuuid*.
`pdf_uuid_equal_p`	`uuid_compare`	Function	The function that compares two given UUIDs uses the comparison function of *libuuid*.

Table 3.2: The final mapping between the API of *libuuid* and the temptative API to be offered by the GNU PDF Library's UUID module.

3.3.3 Implementation

Once the mapping between the *libuuid* and the UUID module APIs was clear, the implementation stage began. The first task to do was to introduce the dependency on *libuuid* in the GNU Build System [61] (better known as **Autotools** and basically consisting of Autoconf, Automake and Libtool [20]). Then, the coding of the UUID module was performed.

Introducing dependency with *libuuid*

The GNU PDF Library, as well as most of the GNU software, uses the GNU Build System, also known as **Autotools**, to make *portable* and *self-configuring* software.

A lot of documentation is available for developers about the GNU Build System ([10, 20, 21]). Briefly, it has evolved from the well known `make` tool, which uses the popular `Makefile` files, to a complete dependecy and portability-aware build system including `autoconf`, which generates the `configure` script according to libraries available on the user's computer. The `configure` script, in turn, generates appropriate `Makefile` files. A complete flow diagram of the process is depicted in figure 3.12.

This flow is started in the GNU PDF Library source code directory with the `autogen.sh` script, which calls `autoconf`, `autheader`, `aclocal`, `automake`, `automake` and `libtoolize` where appropriate, in order to update generated configuration files (see `autoreconf` man page with the `man autoreconf` command).

The *libuuid* dependency is introduced in the GNU PDF Library in two steps. First, the GNU Build System is used to specify that the *libuuid* must be located in the user's system as a pre-requisite. Second, the compiler flags needed to find header files and shared libraries in the system are indicated.

Checking for libraries before building The `autoconf` tool reads the contents of the `configure.ac` file, placed in the source code root directory. This file contains all the checks that the GNU Build System must perform (*e.g.* if required libraries are installed in the system) at the time that the `configure` script is being executed. After these checks, the `configure` script is generated according with the cheking results. The file `configure.ac` organizes checkings in the following groups [10]:

1. Programs checks (programs currently installed in the system)
2. Libraries checks (libraries currently installed in the system)
3. Header files checks (header files currently available in the system)
4. Typedefs checks (data types definitions)
5. Structures checks (structures definitions)
6. Compiler's behaviour checks
7. Library functions checks
8. System services checks

The interesting section here is, obviously, the libraries checks. The goal is to successfully indicate to `autoconf` (through the `config.ac` file) that the `configure` script has to check if *libuuid* is installed in the system. If it is, the configuration must end normally (*i.e.* generating appropriate `Makefile` files); otherwise, if the `configure` script is not able to find the *libuuid* library installed in the system, `configure` has to show a message to the user noticing so. Hence, moving into the libraries checks section, the lines showed in figure 3.13 were added in the `configure.ac` of the working branch.

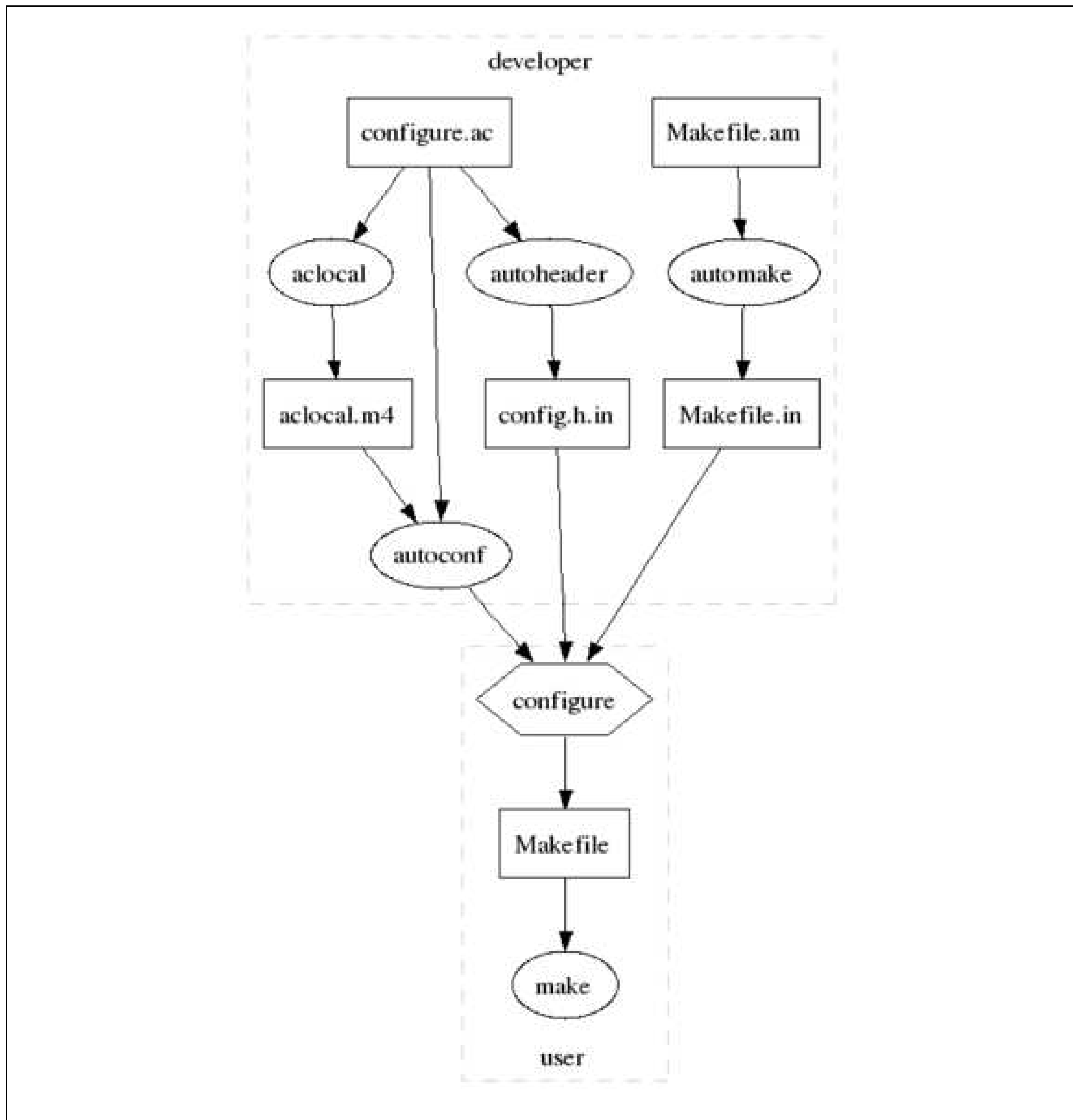


Figure 3.12: Flow diagram including configure, autoconf and automake, three tools in the GNU Build System (image created by Stefan "Polluks" Haubenthal).

These lines are almost self-explanatory. They are divided in two blocks: one specifying the library check, and another telling to `autoconf` what to do if the library is not found.

In the first block, the `AC_LIB_HAVE_LINKFLAGS` macro [33] tells `autoconf` to look up the *libuuid* installation in the default linker path (a system dependent list of directories, that usually contains the `/usr/lib` directory).

In the second block, a variable containing the checking result is queried. If the library was found, then nothing happens. Otherwise, the name of the library (*libuuid*) is appended to a variable containing all the libraries that are missing in the system. This variable is later used to show the corresponding error messages to the user (if any) during the execution of

```
dnl libuuid
AC_LIB_HAVE_LINKFLAGS([uuid])

...

if test "x$HAVE_LIBUUID" != "xyes"; then
   missing_libs="$missing_libs
   libuuid"
fi
```

Figure 3.13: Lines added in configure.ac, requiring the system where the GNU PDF Library is being built to have installed the *libuuid* library.

the `configure` script.

Specifying linker flags Once the build system has been warned about checking if *libuuid* is present or not, there is still another task to do: guarantee that the compiler uses the appropriate header files and linker flags when trying to build the GNU PDF Library (specially, when trying to build the UUID module).

This can be indicated using the `src/Makefile.am` file of the build system. Variables defined in this file are used to generate the appropriate compiler invocation. What is inteded here is that the build calls to `gcc` use the correct values for the options `-I` (*e.g.* `-I/usr/include/uuid/uuid.h`) and `-l` (*e.g.* `luuid`), in order to make the compiler find header files and shared libraries, respectivelly. The changes that were needed to achieve this are shown in figure 3.14.

```
libgnupdf_la_LDFLAGS = $(top_builddir)/lib/libgnu.la \
                       $(LTLIBPTHREAD) $(LTLIBM) \
                       $(LTLIBUUID) $(LTLIBPTHREAD) $(LTLIBM) \
                       $(LTLIBJBIG2DEC) $(LTLIBJPEG) \
                       $(LTLIBCURL) $(LIBGCRYPT_LIBS) \
                       $(LTLIBGPG_ERROR) $(LTLIBGCRYPT) \
                       $(LTLIBCHECK) $(LTLIBICONV)
```

Figure 3.14: Lines added in `src/Makefile.am`, indicating the flags that have to be passed to the compiler to consider shared libraries.

As can be seen in the third line, the *libuuid* flags were added at this point to get the GNU Build System ready to work.

Additionally, some statements had to be added in `src/Makefile.am` to notice the GNU Build System that the UUID module files had to be compiled with the rest of the library. Moreover, the supplied API should be exported within the rest of the public API of the library:

```
TYPES_MODULE_SOURCES = base/pdf-types.h \
                       base/pdf-types-buffer.c base/pdf-types-buffer.h \
                       base/pdf-types-pmon.h \
```

```
                         base/pdf-types-uuid.c base/pdf-types-uuid.h

...

PUBLIC_HDRS = ...
              base/pdf-types-uuid.h \
              ...
```

UUID Module coding

At this point, an updated branch of code with a GNU Build System linking with *libuuid* was ready. The next step consisted of coding a thin layer between the GNU PDF Library and the *libuuid* library. This layer had to implement the required functionalities of the UUID module, using calls to the *libuuid* API (provided functions and data types in `/usr/include/uuid/uuid.h`) when necessary.

The UUID module consists of two files: `src/base/pdf-types-uuid.h` and `src/base/pdf-types-uuid.c`. These names follow GNU PDF Library conventions. All source code files must begin with the `pdf-` prefix, followed by the module where they belong (`types-` in this case, because UUIDs are a subclass of basic types, like boolean or numeric) and finally the particular submodule (`uuid`). Source code files must be conveniently placed in the subdirectory representing the layer where they belong (`base/`) in the source code hierarchy. The former file contains the header (API), while the latter contains the implementation.

`pdf-types-uuid.h` There are two interesting blocks of code in the header: the definition of data types, and the function headers (parameters, function names and return types).

According with functional requirements (see section 3.3.1), the UUID module has to provide two data types: an enumeration of all types of UUIDs that can be generated by the library, and a data type holding the UUID itself.

The enumeration defines two types of UUID: `PDF_UUID_TIME` and `PDF_UUID_RANDOM`. The former refers to time-based UUIDs, while the latter refers to random-based UUIDs. As discussed in 3.3.1, this is not aligned with starting requirements, which stated that time-based, random-based and name-based UUIDs were necessary. The first reason for not including the name-based was that this type of UUIDs are not supported by *libuuid*. Before deciding to implement name-based UUIDs (and where exactly, since the most reasonable option was to improve *libuuid* out of the collaboration with GNU PDF, and then return back to GNU PDF and link with the new functionality), the convenience of name-based UUIDs from the requirements point of view was discussed in `pdf-devel`, as shown in figures 3.15 and 3.16.

The result of the discussion was to drop the name-based UUIDs requirement from the functional requirements list, as the main use of the UUID module generation function would be almost always on time-based UUIDs.

The other data type provided in this header is the `pdf_uuid_t` type, which in this first iteration looked as shown in figure 3.17.

In this first version of the UUID data type, the defining struct contained the `uuid_t` data type defined in *libuuid*, the type that was used to generate it, and a buffer to hold an ASCII representation. This struct, named `pdf_uuid_s`, must be kept away from the visibility the user of the library, who does not need to know how this abstract data type (ADT) is implemented. To achieve this opaqueness on data type definitions, the `typedef` statement

```
Hi,

Regarding this, I have the linkage with Tso's libuuid almost done in
base/pdf-types-uuid.[ch].

However, Tso's libuuid does not provide generation of name-based
UUIDs, as required in doc/gnupdf.texi; it provides time-based and
random-based generation only. So, my question is: is the name-based
generation a must? If it is, I could implement it in libuuid and then
come back to libgnupdf to complete the linkage. If it is not, this
task is almost done :)

By the way, I'm feeling curious about the UUID module: where is it
going to be used?

Albert
```

Figure 3.15: This mail sent to `pdf-devel` exposed that *libuuid* lacked of a name-based UUIDs implementation. If name-based were found to be necessary, a solution was proposed.

defines the `pdf_uuid_t` as a pointer to the `pdf_uuid_s` struct, thus hiding this last to the user manipulating variables of type `pdf_uuid_t` [27]. In this case, the type definition involving a pointer makes the ADT to be dynamically allocated in the heap.

Finally, the function headers were written according to the API previously defined in the GNU PDF Library Reference Manual [28].

`pdf-types-uuid.c` This file covers the implementation of the three following functions:

- `pdf_uuid_generate`
- `pdf_uuid_string`
- `pdf_uuid_equal_p`

Each function calls its equivalent in *libuuid*, making data types conversion to fit GNU PDF requirements when necessary. Dynamic memory is also allocated where it is needed. Some semantic mismatches (*e.g.* the returning types of the UUID comparison function in *libuuid* and the UUID module) get aligned conveniently.

3.3.4 Testing

A tester program was written to check the behaviour of the library at this point. This program had the following goals:

- To include the GNU PDF Library API and check its availability in the developing system.
- To use data types and functions of this API.

```
Hi Albert.

   Regarding this, I have the linkage with Tso's libuuid almost done in
   base/pdf-types-uuid.[ch].

Great :)

   However, Tso's libuuid does not provide generation of name-based
   UUIDs, as required in doc/gnupdf.texi; it provides time-based and
   random-based generation only. So, my question is: is the name-based
   generation a must? If it is, I could implement it in libuuid and then
   come back to libgnupdf to complete the linkage. If it is not, this
   task is almost done :)

We can live without the name-based UUIDs.  We will be using time-based
UUIDs most the time, I guess.

Please send your branch with bzr send when you feel it is ready for
review.

   By the way, I'm feeling curious about the UUID module: where is it
   going to be used?

We will be using it to generate IDs for the PDF documents, as well as in
any occasion where we may need an unique identifier.

Thanks!
```

Figure 3.16: This mail was the response from the list about convenience of name-based UUIDs. `jemarch` exposed that the main use of the UUID module would be generating time-based UUIDs, so the implementation of name-based UUIDs was discarded at this point.

- To check if UUIDs were correctly generated.
- To check if UUIDs were correctly represented in the screen through its ASCII representation.
- To check if UUIDs could be compared between them.

The source code of this testing program can be found online [46]. All the goals were achieved.

3.3.5 First patch

The first patch of the UUID module was submitted to the `pdf-devel` mailing list for discussion on 03/03/2011. A copy of the file `patch-amp-2011-03-03.diff` can be found online in `pdf-devel` archives [30].

```
/* UUID data type */
struct pdf_uuid_s
{
  uuid_t uuid;
  enum pdf_uuid_type_e type;
  char ascii_rep[PDF_UUID_CHAR_LENGTH];
};

...

typedef struct pdf_uuid_s *pdf_uuid_t;
```

Figure 3.17: A first version of the UUID data type. The struct contains the `uuid_t` data type defined in *libuuid*, the type that was used to generate it, and a buffer to hold the ASCII representation of it.

3.3.6 Review

After submitting the patch to the `pdf-devel` mailing list, some observations were made (see figures 3.18 and 3.19).

```
Hi Albert.

   Please find attached a proposed patch

     bzr send -o ./patch-amp-2011-03-03

   implementing the required UUID functionalities.

If the implementation is going to keep the pdf_uuid_s structure in the
heap then I think it would be better to change the name from
pdf_uuid_generate to pdf_uuid_new and to provide a pdf_uuid_destroy
function as well.

Apart from that, it looks good.
```

Figure 3.18: Comments on first UUID module patch. `jemarch` pointed that storing the `pdf_uuid_t` type in the heap implies providing both a creation and a destruction function, in order to allocate and free memory resources when dealing with UUIDs.

All these observations were taken into account to improve the UUID module implementation, and were also used as a starting point in the second development iteration.

3.4 Second iteration

The goal of this iteration was to improve the code sent in the first patch, focusing in:

- Reducing the UUID data type struct to just hold the UUID data.

```
Regarding the pdf_uuid_t type, looking at the API of libuuid and the API
we want to export in pdf.h it looks like we don't need to allocate space
in the heap at all.  It would be better to just make pdf_uuid_t big
enough to hold uuid_t and use uuid_unparse to return the ascii
representation of the uuid.
```

Figure 3.19: Comments on first UUID module patch. The conclusion here is that there is no good reason to store the UUID data type in the heap, since fields defined in the struct are not strictly necessary. Storing only the UUID data and using the stack instead of the heap may lead to less memory consumption.

- Storing the `pdf_uuid_t` in the stack, instead of doing so in the heap, to reduce memory fragmentation.

Some theoretical approaches were studied for achieving these goals [54, 27]. GNU PDF Library criteria regarding the use of the stack or the heap to store program objects (ADT instance) is simple [27]:

- Use **a pointer to a structure** to hold the private data, and then a `typedef` that defines a pointer to that structre (just as done in first submitted patch). This is indicated on complex or big enough data types.
- Use **a structure** (not a pointer to it) to represent the ADT. This alternative is indicated in the case where the private data of the ADT is small, allowing the developer to allocate instances of the ADT in the stack (C structures are passed by value by default when used as function parameters or return types) and thus avoiding fragmentation of the heap.

As required by the reviewing process, the code of the UUID module could be improved by just taking that second approach.

No major changes were considered to be necessary in *requirements analysis* nor *specification and design* development stages. Hence, improvements were introduced by just going straight ahead in the *implementation* phase.

3.4.1 Implementation

No additional changes were needed in the GNU Build System, as the *libuuid* library was successfully linked with the GNU PDF Library in the previous iteration.

UUID Module coding

Improvements were made in both `pdf-types-uuid.h` and `pdf-types-uuid.c` files.

`pdf-types-uuid.h` The most significant improvements in the UUID module header were:

1. To reduce the `pdf_uuid_t` data type, making it just big enough to hold *libuuid*'s `uuid_t` data type.
2. To change the storing of the `pdf_uuid_t` data type from the heap to the stack.

As shown in figure 3.20, these were achieved by:

1. `enum pdf_uuid_type_e type` and `char ascii_rep[PDF_UUID_CHAR_LENGTH]` fields removal from the `pdf_uuid_s` struct definition.

2. Changing the `typedef` statement that defines the `pdf_uuid_t` type to alias `pdf_uuid_s`, instead of holding a pointer to it.

```
struct pdf_uuid_s
{
  uuid_t uuid;
};

...

typedef struct pdf_uuid_s pdf_uuid_t;
```

Figure 3.20: Second version of the UUID data type. The struct contains the `uuid_t` data type defined in *libuuid* only. The `pdf_uuid_t` data type just aliases `pdf_uuid_s`, instead of making a pointer to it.

`pdf-types-uuid.c` Some changes in the UUID module implementation file were needed, according to data type changes in the header.

First, since the `pdf_uuid_t` data type was no longer stored in the heap, there was no need to allocate dynamic memory using `pdf_alloc`, and thus this call was removed. The only field of the structure was initialized through the calls to *libuuid* functions, and then a copy of the struct was returned.

Second, the ASCII representation of the UUID was no longer stored in the `pdf_uuid_t` data type. This meant that this string initialization could not be done in `pdf_uuid_t` anymore, and thus the call to `uuid_unparse` was removed from `pdf_uuid_generate` too.

Third, and according with new behaviour of `pdf_uuid_generate`, unparsing of the UUID was passed to `pdf_uuid_string`, which is more reasonable since not all generated UUIDs may require an ASCII representation. But this lead to a new problem: where should the ASCII string (`char ascii_rep[PDF_UUID_CHAR_LENGTH]`) be stored now? The field inside the `pdf_uuid_t` was not available anymore. Allocating static memory inside `pdf_uuid_generate` should not work and, in fact, it did not: as being a local variable, its value is lost when the execution goes outside the function's scope.

Going deep inside this issue [52], three possible solutions were found to be capable of solving the problem, with pros and cons:

1. **Allocating dynamic memory** for the string. This would easily solve the problem, since a call to `pdf_alloc` would return a pointer to a heap region (out of the execution stack) big enough to hold the ASCII representation. This pointer could be returned to the user, and would still be valid although the execution scope changed. However, the heap is being used again, and hence memory fragmentation would increase using this solution.

2. **Making the string `static`**. This avoids the array to disappear when `pdf_uuid_string` returns, so the poi nter is still valid by the time the caller uses it. Each time the function is called, it re-uses the same array and returns the same pointer. This solution works well, but it has some difficulties that will be discussed later.

3. **Let the user allocate an static array** big enough, and call `pdf_uuid_string` with it as a parameter. This, obviously, would change the form of the `pdf_uuid_string` header defined in the specification. It also requires the user to know the char array size to allocate, which is something uncomfortable.

First and last options were discarded in this iteration. The first option, allocating dynamic memory, was found again to increase memory fragmentation (the char array hold just 46 bytes, which is a very little portion of data to use the heap). The last option had two consequences. First, it required to change the specification's function header, which was something avoided as much as possible. Second, it implied the user to know how much memory is needed to store an ASCII representation of a UUID, which was found unclear and error prone. All these lead to choose the second option, making the char array `static`, and this was how finally it was coded in the second proposal patch.

3.4.2 Testing

The same tools of the first iteration were used to check the improvements made in this one. All functionalities of the UUID module were working successufully whith the new changes.

3.4.3 Second patch

The second patch of the UUID module was submitted to the `pdf-devel` mailing list for discussion on 15/03/2011. A copy of the file `patch-amp-2011-03-15.diff` can be found online in `pdf-devel` archives [30].

3.4.4 Review

With the usual procedure, the patch was reviewed by the GNU PDF developing community. As expected, the decisions involving the ASCII storing mechanics recieved some observations (see figure 3.21).

The two remaining choices regarding this issue (allocating dynamic memory, and making the user to allocate previsouly a static char buffer) were exposed in reply. An interesting discussion began in the `pdf-devel` mailing list and channel `#pdf` at `irc.freenode.net`, mainly with users `jemarch` and `aleksander_m`.

As `jemarch` pointed out, the main disadvantage of using the `static` modifier on the ASCII buffer was that it made the call to `pdf_uuid_string` non-reentrant [67]. This means that concurrent calls to pdf_uuid_string may overwrite the UUID ASCII data *before* the programmer has the chance to use it. The typical example exposed was using simple inline calls like these:

```
printf("The ASCII value of first UUID is %s,
        while the value of the second is %s\n",
        pdf_uuid_string (uuid1),
        pdf_uuid_string (uuid2));
```

```
Hi Albert.

   Please find attached another patch with your suggested modifications
   (pdf_uuid_t is no longer allocated in the heap).

Looks good.

Just one question.  In the function pdf_uuid_string:

  +const char *
  +pdf_uuid_string (pdf_uuid_t uuid)
  +{
  +  static char ascii_rep[PDF_UUID_CHAR_LENGTH];
  +
  +  uuid_unparse (uuid.uuid, ascii_rep);
  +
  +  return ascii_rep;
  +}

Is there a specific reason to use the static buffer 'ascii_rep' instead
of allocating a new string?  That makes pdf_uuid_string non-reentrant.
```

Figure 3.21: Comments on second UUID module patch. Using the `static` modifier to return always the same array pointer had some problems too.

It is obvious that these inline calls are executed sequentially. The first call to the function `pdf_uuid_string` writes in the `static` buffer the ASCII value of `uuid1`; then, the second call does so for `uuid2`, overwriting the value previously written of `uuid1`, since the `static` buffer is the same for all `pdf_uuid_string` calls. As these calls return always the same memory address, the strings that `printf` outputs in `stdout` are exactly the same for both UUIDs, and its value is the ASCII data of `uuid2` (which wrote the `static` buffer for the last time). This behaviour is contrary to what is expected by the programmer, and was considered unacceptable.

3.5 Third iteration

The goal of this iteration was to change the design of the `pdf_uuid_string` function to fit the reviewing of the `pdf-devel` mailing list. As stated above, the implementation of the character array holding the ASCII representation of a UUID presented three options:

1. Store it allocating (via `pdf_allocate`) heap space.
2. Store it in a `static` defined array.
3. Let the user declare a buffer in its working scope, and pass a reference to it as a `pdf_uuid_string` parameter.

First and second options were discarded by their cons: heap allocating causes memory fragmentation (and its specially a bad option for a little –46 bytes– piece of data), while `static` arrays cause the function to be non-reentrant (discussed above, see section 3.4.4). These issues were considered unacceptable, and then the last option was the community choice.

The third option has some consequences for the programmer using the GNU PDF Library and the UUID module: in the first proposal the ASCII representation was stored internally in a buffer inside the `pdf_uuid_t` data type; now it is a responsibility of the programmer to initialize conveniently a buffer in its local scope and call `pdf_uuid_string` with a reference to this buffer. Efficiency is the main advantage of this choice, since it requires less resources by the GNU PDF Library. On the other hand, library users have now to initialize their ASCII buffers appropriatelly, and then call `pdf_uuid_string`.

3.5.1 Implementation

As well as in the second iteration, no additional changes were needed in the GNU Build System or the rest of the library configuration.

UUID Module coding

To change the implementation of the `pdf_uuid_string` and the buffer it uses, some improvements had to be made in both `pdf-types-uuid.h` and `pdf-types-uuid.c`.

`pdf-types-uuid.h` Changes in the UUID module header were totally related to the function `pdf_uuid_string` and the storing of the UUID ASCII buffer. Two issues had to be fixed here: first, the function header should require extra parameters to specify a pointer to a previously initialized buffer (see figure 3.22). Second, a private constant should define the minimum length of such a buffer, to guarantee that the pointer passed to `pdf_uuid_string` is long enough to hold an ASCII representation of a UUID (see figure 3.23).

```
/* Printed ASCII representation of an UUID */
pdf_char_t * pdf_uuid_string (pdf_uuid_t uuid,
                              pdf_char_t * buffer,
                              pdf_size_t buffer_size);
```

Figure 3.22: Detail of the extra parameters needed in `pdf_uuid_string`. First parameter was previoiusly included, and requires a UUID instance. Second parameter is a pointer to a previously initialized buffer. Finally, third parameter requires the user to specify the size of the buffer referenced in the second one.

```
#define PDF_UUID_SIZE 46
```

Figure 3.23: Detail of the UUID ASCII size constant statement. This is the recommended minimum number of octets to hold an entire representation of a UUID [28].

It is important to note that the constant described in figure 3.23 is *private*, and thus it is not provided to programmers through the API exported during the build process. Hence,

the requirement to pass a buffer having a length greater or equal than this constant's value was conveniently described in the GNU PDF Library documentation file (`gnupdf.texi`).

All documentation contributions were also included in the submitted patch (available online [30]).

`pdf-types-uuid.c` The implementation to achieve the goals for this iteration was quite simple. First, the constant previously defined in the header `pdf-types-uuid.h` was used in the very first lines of the function to verify if the passed buffer had enough length to hold a UUID ASCII string. If it has not, then the `NULL` pointer is returned. Then, the pointer to the buffer previously initialized by the user has to be passed to *libuuid*'s `uuid_unparse` to hold the ASCII representation.

Some issues raised during the coding of this file. The most remarkable follow:

1. First modifications of the UUID module header presented an ambiguous parameter specification regarding the buffer pointer that has to be passed to `pdf_uuid_string`. This led to some pointer mess. Specifically, char array pointers were being passed to `pdf_uuid_string` without pre-allocating space at all. This obviously caused segmentation faults when trying to get ASCII strings for UUIDs.

2. Since it was syntactically correct, some buffers were passed to `pdf_uuid_string` pre-allocating less space than needed (*i.e.* less than 46 octets). This caused buffer overflows, and thus stack corruptions. Since undefined behaviour happens when the stack is corrupted with a buffer overflow. The C language has no internal mechanisms to avoid buffer overflow. What makes this sort of error difficult to debug is that it casuses undefined behaviour. Undefined behavior means behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the International Standard imposes no requirements. This means that possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). This problem, then, was certainly difficult to debug. Tools like Valgrind [8] and others [57] were used to determine what was going wrong.

3. The returning type for the `pdf_uuid_string` function was first `char`. The community asked to change this to `pdf_char_t`, since the library has its own data type for managing characters (and the rest of basic data types typically provided by the language) to ensure proper portability.

The first two issues were finally solved introducing a third parameter to the function `pdf_uuid_string`, asking the caller to specify the exact size of the buffer indicated in the second parameter. Proceeding this way, there was no possibility of overflowing this buffer: if the user specifies a size less than the `PDF_UUID_SIZE` constant, then the `NULL` pointer is returned. This way, when the parameter is invalid the result returned by the function is invalid too. The user is required to read the requirements for the function and to initialize the buffer appropriately if he wants to get a correct ASCII representation of his or her UUIDs.

3.5.2 Testing

The testing program of previous patches was conveniently modified to fit with new function headers. All functionalities continued to work properly, and new ones, such as the thread-safe version of `pdf_uuid_string`, ran successfully.

3.5.3 Third patch

The third patch of the UUID module was submitted to the `pdf-devel` mailing list for discussion on 24/03/2011. A copy of the file `patch-amp-2011-03-24.diff` can be found online in `pdf-devel` archives [30].

3.5.4 Review

The third patch passed the reviewing process, and was applied to the main branch of the GNU PDF Library (see figure 3.24). This means that further updates and downloads of the library source code will include the UUID module implementation, freely available to everyone who needs to use it or wants to improve it.

```
Hi Albert.

   Ok, issues fixed and some improvements added (now a pdf_char_t is
   returned by pdf_uuid_string to allow inline calls; added a check for
   the appropriate buffer size; improved pdf_uuid_generate &
   pdf_uuid_compare code; updated gnupdf.texi)

Applied.  Thanks for the patch.

By definition it is not easy to predict what a newly generated uuid will
be, but may be useful to have some simple tests running the generation
function and checking the structure of the printed representation.
```

Figure 3.24: Third patch acceptance notification from `jemarch`. The patch was applied to the main branch of the GNU PDF Library, and thus making it freely available for everybody to use and improve.

As the response remarks too, some unit tests integrated in the GNU PDF Library testing mechanism may be useful, and would make the UUID module more consistent.

3.6 Fourth iteration

This iteration consisted of developing a series of unit tests for the UUID module within the GNU PDF Library testing framework. The goals for this iteration were:

1. To learn how the GNU PDF Library testing infrastructure works.
2. To provide tests checking the functionalities provided by the UUID module.

The following section explains how the GNU PDF Library testing infrastructure works, and thus tries to achieve the first goal. Unit tests and test cases designed for this iteration are then explained.

3.6.1 GNU PDF Library Testing Infrastructure

The GNU PDF Library Hackers Guide [27] explains how the testing environment for the library works.

The development group is following a bottom-up testing strategy. The verification of the library is performed in the following steps (see figure 3.25):

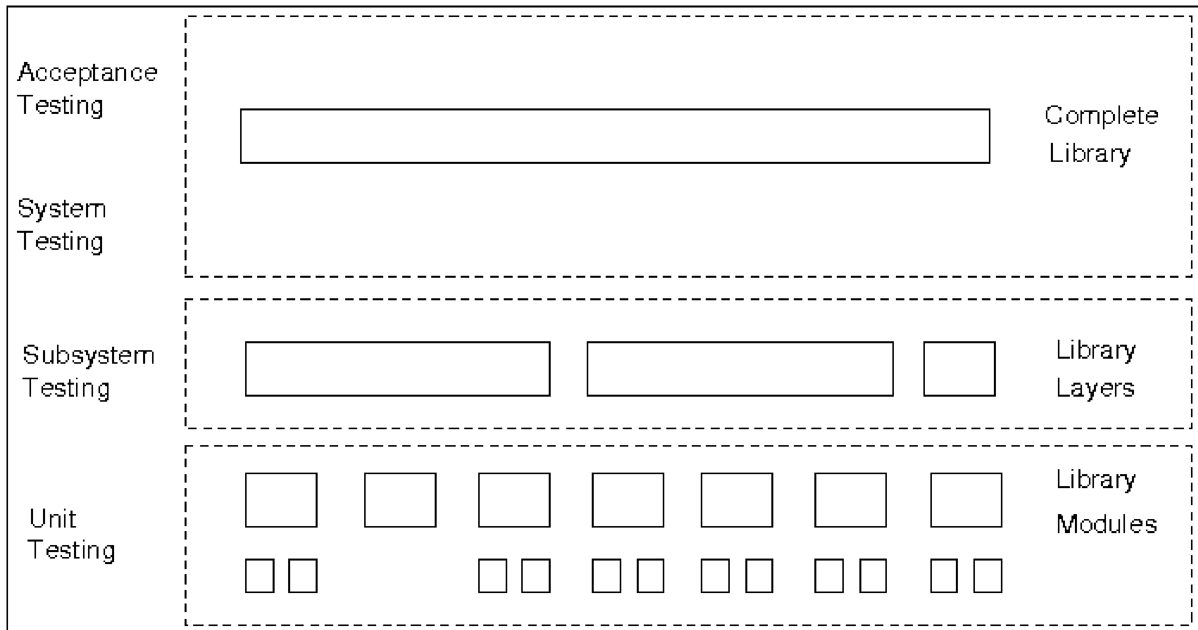


Figure 3.25: Library testing strategy. The GNU PDF Library is tested under several scopes.

- **Unit testing** is performed in order to verify the low-level modules of the library.
- **Subsystem testing** is performed in order to verify the combination of several subsystems, *i.e.* to test each library layer.
- **System testing** is performed in order to verify the whole system. i.e. the GNU PDF Library.

The library uses *libcheck* [7] to implement the testing infrastructure. A testing report is automatically updated daily from the main branch, launching all implemented tests. This report is available online [29].

Unit tests are organized as shown in figure 3.26. Test suites are used to collect unit tests for a given module. In turn each test suite contain a collection of test cases. Each test case identifies a function implemented in the module. Several tests can then be defined to test the function capabilities.

As shown in figure 3.27, the unit tests are stored in the `torture/unit/` directory.

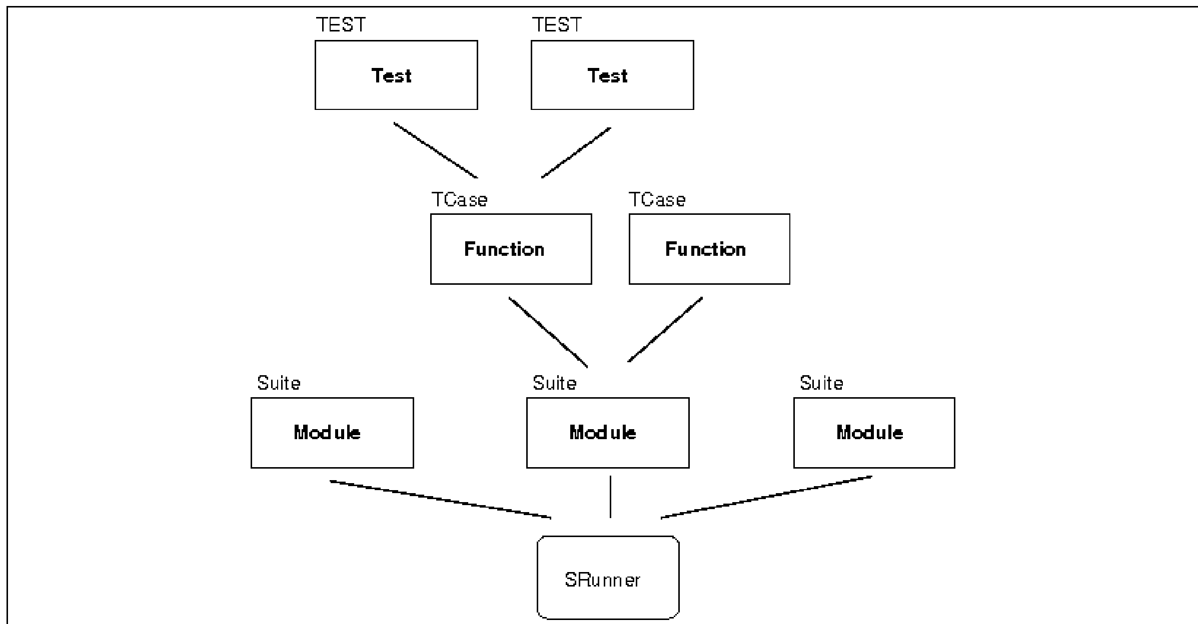


Figure 3.26: Unit testing architecture.

3.6.2 Testing

This section describes the tests written for each UUID module function. The source code of all tests implemented is avaiable online in `pdf-devel` archives [30].

`pdf-types-uuid-generate.c`

This file contains tests against the `pdf_uuid_generate` function, which generates UUIDs for the library. The goal is to generate some UUIDs in a unique test, since the generation cannot fail in the library context. The parameter value cannot be invalid during execution, since it is checked at compile time to have one of the allowed UUID types. In the worst case, it may segfault for an external cause (such total memory consumption), in which case the kernel may kill the process. In that chase, the *libcheck* library informs correctly about what caused the crash.

Test case: `pdf_uuid_generate`

1. `pdf_uuid_generate_001`
   - **Test**: `pdf_uuid_generate_001`
   - **Description**: Generate some UUIDs of supported types.
   - **Success condition**: Generated UUIDs should be ok.

`pdf-types-uuid-string.c`

This file contains tests against the `pdf_uuid_string` function, which returns an ASCII representation of a given UUID. The goal is to check if everything goes well in the following environments:

- The external buffer holding the ASCII string is allocated in dynamic memory (heap)

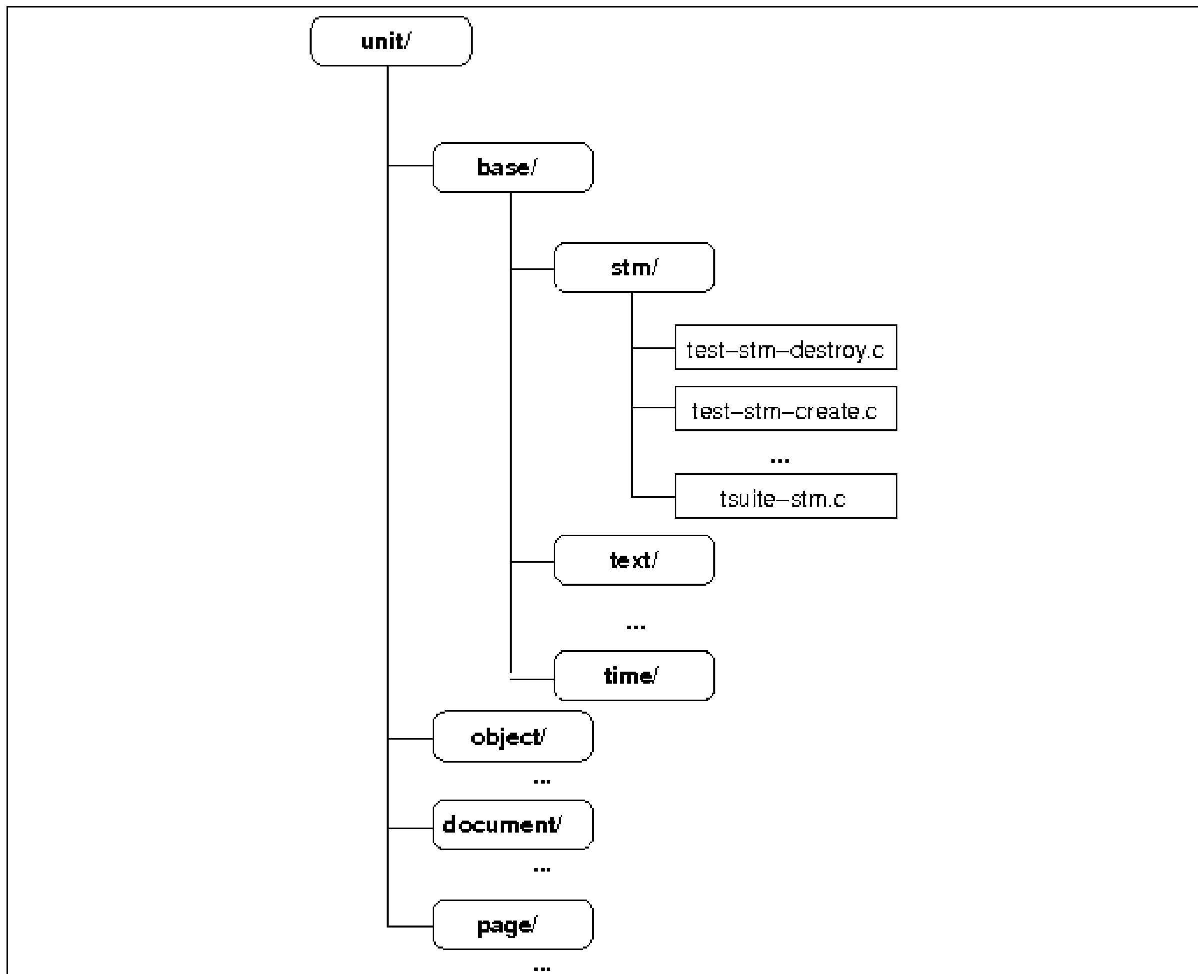


Figure 3.27: Unit testing sources.

- The external buffer holding the ASCII string is allocated in static memory (stack)
- The external buffer holding the ASCII string is very big
- The internal structure of the generated ASCII string meets the UUID representation standard

The test case briefing follows.

Test case: `pdf_uuid_string`

1. `pdf_uuid_string_001`
   - **Test**: `pdf_uuid_string_001`
   - **Description**: Generate an UUID ascii representation in heap.
   - **Success condition**: Generated UUID ascii should be ok.
2. `pdf_uuid_string_002`

- **Test**: `pdf_uuid_string_002`
- **Description**: Generate an UUID ascii representation in stack.
- **Success condition**: Generated UUID ascii should be ok.

3. `pdf_uuid_string_003`
   - **Test**: `pdf_uuid_string_003`
   - **Description**: Generate an UUID ascii representation, buffer size greater than required.
   - **Success condition**: The buffer to store the UUID string is long enough. The conversion should pass.

4. `pdf_uuid_string_004`
   - **Test**: `pdf_uuid_string_004`
   - **Description**: Cheks for appropriate structure of UUID ascii.
   - **Success condition**: The ascii structure is 00000000-0000-0000-0000-000000000000, where 0s match hexadecimal digits.

`pdf-types-uuid-equal-p.c`

This file contains tests against the `pdf_uuid_equal_p` function, which compares two given UUIDs and determines if they are equal or not. The goal is to generate a simple pair of tests checking one of the most important UUID properties: if two nearly generated UUIDs collide or not.

Test case: `pdf_uuid_equal_p`

1. `pdf_uuid_equal_p_001`
   - **Test**: `pdf_uuid_equal_p_001`
   - **Description**: Checks if two time-based UUIDs are equal.
   - **Success condition**: Both time-based UUIDs are different.

2. `pdf_uuid_equal_p_002`
   - **Test**: `pdf_uuid_equal_p_002`
   - **Description**: Checks if two random-based UUIDs are equal.
   - **Success condition**: Both random-based UUIDs are different.

3.6.3 Testing environment execution

The build system allows to compile and execute only the desired tests (for example, one test suite alone executing tests for a unique library module). Issuing the following command from the directory source root:

```
$ make check MODULE=types
```

Compiles and executes the test framework for the Basic Types module only, which makes sense here because, at the implementation moment, the only tests performed in this module were those belonging to the UUID submodule. The results were the following:

```
Running suite(s): types
100%: Checks: 7, Failures: 0, Errors: 0
PASS: runtests.sh
```

3.6.4 Fourth patch

The fourth patch of the UUID module was submitted to the `pdf-devel` mailing list for discussion on 30/05/2011. A copy of the file `patch-amp-2011-05-30.diff` can be found online in `pdf-devel` archives [30].

3.6.5 Review

The fourth patch passed the reviewing process on 19/06/2011. Provided tests are executed automatically among the rest of the testing infrastructure during automatic reports [29].

Chapter 4

PDF standards requirements

One of the most important desired characteristics for the GNU PDF Library, as discussed in chapter 2, is to be *complete*. Completeness means, in this project, supporting the full PDF specifications and standards as published by Adobe and the ISO, respectivelly.

Conformance with official PDF specifications and standards is one of the main goals of the GNU PDF project. This goal distinguishes the GNU PDF Library from the rest of libraries and programs that bring PDF technology to the users (excepting those from Adobe). These libraries and programs are not specificallly written to support these specifications and standards; they go straight ahead to allow users to read and write PDF documents. Moreover, the GNU PDF Library wants to be a certified library, becoming in the mid/long term a true and free alternative to Adobe SDK.

The chapter is organized as follows. First, an introduction describes how PDF standards are involved in several layers of the GNU PDF Library. Second, a glossary of important concepts for this chapter is listed and explained. Third, an analysis of the set of all available PDF standards is performed. Fourth, a comparison of the requirements between some of these standards is described. And finally, the importance of a conformance module is argued, and a conformance API consisting of submodules, data types and functions is proposed.

4.1 Introduction

Since 1993, Adobe Systems Inc. has released a number of specifications for the PDF format. These specifications contain a set of requirements that define the format itself, and a deep study of them (see section 4.4) shows that these requirements belong usually to one of the following groups:

1. **Syntactic** requirements. These requirements describe the formal PDF language, thus establishing implicitly concrete syntax rules (*i.e.* a PDF grammar). This allows a criterion to say if a PDF file is valid or invalid, correct or incorrect, from a syntactic point of view.

2. **Semantic** requirements. Though being syntactically correct (*i.e.* belonging to the language recognized by the PDF grammar), a PDF file may contain some unacceptable contents. For instance, a PDF objct may reference another PDF object which does not exist in the document. Or a PDF stream object may include a compressed image, without specifying encode/decode parameters appropriately.

3. **Reader** requirements. Programs that read PDF files may have additional requirements on how they treat values or display page contents.

4. **Writer** requirements. Programs that write PDF files may have additional requirements on how they create or update components of PDF files.

As shown, these requirements differ from those expected from, for instance, a programming language. While some semantic errors may be comparable between PDF and programming languages (an indirect reference to a non-existing PDF object is something similar to a null pointer exception) a lot of semantic requirements are related to *media* (fonts, image codecs, and so on).

Moreover, the PDF format is too extensive to many daily tasks. It includes lots of features that most users rarely may need. This reason led the International Organization for Standardization (ISO) to promote PDF standards that use a subset of the complete feature set to achieve the most common goals. This PDF subsets are described in section 4.3.

The requirements gathered by these specifications and standards are decisive in the lexical, syntactical and semantical analysis of PDF files. This means that a lexical processor (*i.e.* a **tokeniser**), a syntactic analyser (*i.e.* a **parser**) and a semantic engine are subject to these requirements contents.

Several documents concerning PDF specifications were previously studied in GNU PDF, specially ISO 32000-1:2008 [2], to appropriately implement a number of modules of the base layer. In addition, a **tokeniser** was already working when the development of this FDP began. The study of standards, specifications and conformance had to be done before making any effort on syntactic or semantic analysis of PDF files. Parsing PDF files was considered a major milestone by the GNU PDF project.

4.2 Concept glossary

This chapter makes an effort to describe the PDF technology analyzing the current state of the art in PDF specifications, standards and other documentation. Before going deep into it, it has been considered necessary to present a list of concepts and terms that will appear along the following sections.

Definition 1. Standard. *The term* standard *is used in this document to refer to an official and published document by the ISO.*

Definition 2. Specification. *Adobe has released over the years a number of documents containing a full specification of the different base versions of PDF (from PDF 1.1 to PDF 1.7). This document refers to these documents as* PDF specifications. *ISO 32000-1:2008 [2] is the only overlapping case between a* standard *and a* specification. *On one hand, Adobe has its own specification document and, on the other hand, the ISO has released its own standard. However, both documents are totally aligned.*

Definition 3. Subset. *As presented in section 4.3, the PDF specifications are restricted by some standards to achieve specific goals (*i.e. *generating PDF for long term preservation). The restrictions in these standards conform a PDF* subset. *PDF/A, PDF/X or PDF/E are typical examples of these subsets. Usually there is not a one-on-one correspondence between a* subset *and a* standard*: sometimes, more than one standard is used to define a single subset. However, given a standard, it defines only one subset restricting only one PDF specification.*

Definition 4. Conformance level. *Alothough a standard refers to only one subset and restricts only one PDF specification, it may define several conformance levels. For example, ISO 19005-1:2005 restricts PDF 1.4 to define two conformance levels: PDF/A-1a and PDF/A-1b. Some restrictions apply to both, while some others apply to only one of them. Hence, it is not correct to say that a PDF file is PDF/A conformant; a PDF file can only be confomant with one conformance level, and thus it should be said that the file is PDF/A-1a or PDF/A-1b conformant.*

Definition 5. Requirement. *A* requirement *is a singular documented need of what a particular product or service should be or perform [68]. PDF requirements are described in paragraphs, inside subsections and sections of standards and specifications.*

Definition 6. Restriction. *In this document, the term* restriction *is used in the scope of PDF standards that reduce the whole set of PDF capabilities described in PDF specifications. Hence, a restriction is a special type of requirement, that appears in PDF standards and helps to define a PDF subset by disallowing or forbidding features present in the global specification.*

Definition 7. Capability. *This term is used in this document as a synonym for* feature *or* functionality. *Capabilities describe precisely what a given PDF version or subset can do, like displaying video on annotations or using JavaScript to validate forms.*

4.3 PDF standards

The whole functionality set provided by the PDF format is contained in the PDF 1.7 specification (ISO 32000-1:2008 [2]). However, some of these functionalities are not strictly necessary in all real world scenarios. For example, PDF 1.7 specifies how a PDF file may contain movies, sounds and other media; but these capabilities make no sense for, let's say, a storekeeper who wants to store all the invoices of his or her business in a digital format (this saves trees, energy, and makes data more persistent!). Thus, availability of PDF features varies depending on the purpose the PDF file exists for.

Specialized subsets of PDF gather those purpose-based specific requirements. Some of them have been standardized, which means that an official published ISO standard describes what the subset stands for and its peculiarities; some other, however, are still in a standardization process.

Through years, many complete PDF specifications have been released, from 1.0 (1993) to 1.7 (though not all of them have been standardized). On one hand, each subset of PDF is based on a whole PDF specification (for example, PDF/A is based on PDF 1.4). On the other hand, PDF 1.7, which is the most recent full standardized specification of PDF, includes all of the functionality previously documented in PDF specifications for versions 1.0 through 1.6. Hence, PDF 1.7 (documented in ISO 32000-1:2008) is taken in this article as a basis to compare all specialized subsets of PDF against it. This means that, despite of being based on PDF 1.4, requirements of PDF/A are compared here with ISO 32000-1:2008 specification.

Note, additionally, that Adobe applies some extensions to ISO 32000-1:2008 which usually are embraced in the name PDF 1.7. All references to PDF 1.7 in this article exclude these extensions. Hence, note that all references to PDF 1.7 in this article refer exclusively to the specification contained in ISO 32000-1:2008.

4.3.1 Standardized subsets of PDF

This subsection describes the current state of the art in PDF standardized subsets.

PDF/X

PDF/X (standardized 2001) stands for *PDF for eXchange*. It offers a specification of PDF aimed at **printing**, **graphic arts** and **prepress digital data**[66].

PDF/X is formalized in ISO 15929 and ISO 15930. However, ISO 15929 was withdrawn in 2008 and is no longer an official standard for PDF/X. **ISO 15930** defines different subtypes or conformance levels for PDF/X. They are described in table 4.1[65].

Specific subtype	**Standard**	**Features allowed**	**Based on PDF**
PDF/X-1a:2001	ISO 15930-1:2001	Blind exchange in CMYK; Spot Colors	1.4
PDF/X-2	ISO 15930-2	Never published	N/A
PDF/X-3:2002	ISO 15930-3:2002	CMYK; Spot Calibrated (managed) RGB CIELAB, with ICC Profile	1.3
PDF/X-1a:2003	ISO 15930-4:2003	Revision of PDF/X-1a:2001	1.4
PDF/X-2	ISO 15930-5	Extension of PDF/X-3; Allows OPI-like data to be included	N/A
PDF/X-3:2003	ISO 15930-6:2003	Revision of PDF/X-3:2002	1.4
PDF/X-4	ISO 15930-7:2008 and ISO 15930-7:2010 (minor corrections and improvements)	Color-managed, CMYK, grayscale, RGB, Spot color data supported; Transparency; There is a second level conformance (PDF/X-4p) if ICC Profile is externally supplied	1.6
PDF/X-5	ISO 15930-8:2008 and ISO 15930-8:2010 (minor corrections and improvements)	Conformance level PDF/X-5g (extension of PDF/X-4): enables use of external graphical content; Conformance level PDF/X-5pg (extension of PDF/X-4p): enables use of external graphical content in conjunction with a reference to an external ICC Profile Conformance level PDF/X-5n (extension of PDF/X-4p): enables externally supplied ICC Profile to use a colorspace different than grayscale, RGB or CMYK	1.6

Table 4.1: The ISO 15930 (PDF/X) series and conformance levels.

PDF/A

PDF/A (standardized 2005) stands for *PDF for Archive.* It offers a specification of PDF aimed at **long term preservation of documents**.

PDF/A is formalized in **ISO 19005 series** [11]. ISO 19005 defines different subtypes or levels of conformance for PDF/A. They are described in table 4.2.

Specific subtype	**Standard**	**Features allowed**	**Based on PDF**
PDF/A-1	ISO 19005-1:2005	Conformance level PDF/A-1a: PDF file must meet all requirements described in ISO 19005 Conformance level PDF/A-1b: PDF file must meet all requirements described in ISO 19005, excepting Unicode character maps requirements and logical structure requirements	1.4

Table 4.2: The ISO 19005 (PDF/A) series and conformance levels.

PDF/E

PDF/E (standardized 2008) stands for *PDF for Engineering.* The standard defines a format for the creation of documents used in engineering workflows. It is designed to be an open and neutral exchange format for engineering and technical documentation [63, 1].

PDF/E is formalized in standard **ISO 24517-1:2008** [12]. Key benefits of PDF/E include:

- Reduces requirements for expensive and proprietary software.
- Lower storage and exchange costs, with respect to paper.
- Trustworthy exchange across multiple applications and platforms.
- Self-contained.
- Cost-effective and accurate means of capturing markups.
- Developed and maintained by the PDF/E ISO committee.

PDF/VT

PDF/VT (standardized 2010) stands for *PDF for exchange of Variable data and Transactional printing.* It defines the use of PDF as an exchange format optimized for Variable Data Printing (VDP) and transactional printing [64]. VDP is a form of on-demand printing in which elements such as text, graphics and images may be changed from one printed piece to the next, without stopping or slowing down the printing process and using information from a database or external files (*i.e.* dynamic contents).

PDF/VT is formalized in standard **ISO 16612-2:2010** [13].

4.3.2 Non-standardized subsets of PDF

PDF/UA

PDF/UA stands for *PDF for Universal Accessibility.* The goal of the development of this subset is to ensure accessibility for people that use assistive technology, such as screen readers for users who are blind.

Although a standard does not exist yet for this subset, a draft is on the way (ISO/DIS 14289-1 [14]).

PDF/H

PDF/H stands for *PDF for Healthcare.* The goal of the development of this subset is to determine a format or container for transfer and storage of health data.

PDF/H is not a standard or proposed standard, but only a best practices guide (BPG) for use with existing standards and other technologies. PDF/H BPG is based on PDF 1.6 [66].

4.4 PDF/A vs PDF 1.4 requirements

This section describes the differences between ISO 19005-1:2005 [11] (from now, referred as *PDF/A* as well), which describes the PDF/A-1a and PDF/A-1b conformance levels, and the non-standardized document *PDF Reference, Third edition* (from now, referred as *PDF 1.4* as well), from Adobe, which describes the PDF 1.4 format, in terms of requirements alignment.

Constraints described in PDF/A restrict the overall capabilities offered by PDF 1.4, in order to ensure that PDF/A compliant files contain features that are stricly needed for archiving purposes only.

Having ISO 19005-1:2005 on one hand, and the PDF Reference Third Edition (PDF 1.4) on the other, requirements alignment between both was studied. For doing so, the sectioning of the first document was considered in first place. Then, for each requirement in a given section, its location in the PDF Reference for PDF 1.4 was searched. Once found, both requirement definitions were compared. As a result, it was stated if both requirements were required, optional, recommended or any other alignment status. If it proceed, an extract of the exact phrase or sentence producing this alignment status was copied.

The results of this comparison can be found in two places. First, as an appendix of this report (see appendix C). And second, as a contribution article to the *PDF Knowledge* base. The PDF Knowledge base is an effort of the GNU PDF project to produce free documentation about the PDF format. The contributed article, which features a comparison table that summarizes the work of this section, can be found online [5].

The following subsections are equivalent to those described in the ISO 19005-1:2005 document [11]. In each subsection, a summary of the comparison work performed is presented, with additional comments when necessary.

Note that only non-aligned requirements are depicted here. This should not happen because PDF subset standards shall constraint whole PDF specifications (*e.g.* PDF/A vs PDF 1.4). This is not true for all requirements contained in PDF subset standards. Often, PDF subset standards contain requirements that are totally aligned with the PDF specification; sometimes, the difference is just a matter of semantics, as both requirements mean exactly the same (*e.g* concepts such as *end-of-line character* and *new line* are confusingly merged). Thus,

PDF subset standards can not, at least formally, be considered as a set of pure restrictions, since they contain both restrictions and redundant, repeated or obvious requirements. These last are not discused in this report.

The following subsections contain briefings, descriptions and overall conclusions about the comparsion performed between both sets of requirements. Please, for a more formal, complete and traceable analysis see appendix C or visit the GNU PDF Knowledge base [5].

4.4.1 File structure

A PDF file has a well defined structure. Essentially, and without going deep into details, it consists of a *header*, a *body* and a *cross-reference table*. The header is a simple comment (or comments) with information about the PDF version or the encoding. The body contains a (usually large) collection of all PDF objects in which the PDF file consists of. Finally, the cross-reference table is a table with offset locations of each PDF object, and it allows to quickly jump between them. However, PDF/A (and rarely any PDF subset) does not impose too much restrictions on this structure, neither the internal tree structure of a document (which relies on the body), and neither does it over PDF general syntax.

The only syntactic requirement for a PDF 1.4 compliant file to be a PDF/A compliant file too, is to have a % character followed by at least four characters, each of whose encoded byte values shall have a decimal value greater than 127. This is to ensure that conforming readers will catch binary content. A lot of extra syntactic requirements are specified too in PDF/A, but those are totally aligned with PDF 1.4 base requirements. Redundancy or overfitting requirements is a very common situation in ISO standards that impose restrictions on a base specification, which is the case.

However, PDF/A imposes several *semantic* requirements on PDF 1.4 files. For instance, the `LZWDecode` filter is not permitted in neither PDF/A-1a or PDF/A-1b compliant files, because LZW compression algorithm is subject to intellectual property constraints. But, apart from this one, most of the semantic requirements of this section come from the essence of PDF/A: archiving purposes. For example, a stream object dictionary can not contain the `F`, `FFilter` or `FDecodeParams` keys. These keys are used to point to document content external to the file. The explicit prohibition of these keys has the implicit effect of disallowing external content that can create external dependencies and complicate preservation efforts. Depending on external files goes against archiving, and thus pointers to external files are disallowed in PDF/A conformant files.

4.4.2 Graphics

Dealing with graphics is always a large part in PDF specifications, and PDF/A restricts many aspects of it. A number of extra requirements is there derived from the following statement: if a user creates a PDF/A conformant file, he or she wants to ensure it for long time preservation. Applying this to *graphics*, the requirement evolves into something like: if the document has to be preserved for a long time, then colours, images and objects have to be specified in a way that their future appearance (rendering) is as close as possible to their present appearance (ISO 19005-1:2005 calls this a *predictable rendering*).

This implies several *semantic* and *reader* specific restricions. For example, the colour characteristics of the device where general PDF 1.4 files are rendered are specified by means of a concrete dictionary, called **OutputIntent**. This dictionary has to be a PDF/A Out-

putIntent, which is a PDF/A restricted OutputIntent that allows to represent colour spaces in a device-independent manner (this can also be achieved using device-independent colour spaces). This enables predictable colour rendering based on a colorimetric definition and without reliance on assumptions or information external to the conforming file. It also provides a mechanism whereby a colorimetric definition can be associated with device-dependent colour data.

Other restrictions apply to images. For instance, the `Interpolate` key must be *false* if present, because reader interpolation may vary along implementations or time, and thus render of the PDF/A file may differ from one reader to another.

Additionally, reference XObjects and PostScript XObjects are forbidden in PDF/A (they can appear in PDF 1.4). The formers refer to arbitrary document content in external PDF files, creating external dependencies that complicate preservation efforts. The latters contain arbitrary executable PostScript code streams that have the potential to interfere with reliable and predictable rendering.

4.4.3 Fonts

The intent of the requirements in this subsection is to ensure that future rendering of the textual content of a conforming PDF/A file matches, on a glyph by glyph basis, the static appearance of the file as originally created, and to allow the recovery of semantic properties for each character of the textual content.

A font is represented in PDF as a dictionary specifying the type of font, its PostScript name, its encoding, and information that can be used to provide a substitute when the font program is not available. The font program itself must be embedded as a stream object in a PDF/A file, with a solely expection.

All PDF/A conforming readers must use the embedded fonts, rather than other locally resident, substituted or simulated fonts, for rendering. This guarantees that fonts will look like the originals did.

4.4.4 Transparency

PDF 1.4 defines a transparency model. Under the transparent imaging model, all of the objects on a page can potentially contribute to the result. Objects at a given point can be thought of as forming a transparency stack (or just stack for short), arranged from bottom to top in the order in which they are specified. The color of the page at each point is determined by combining the colors of all enclosing objects in the stack according to compositing rules defined by the transparency model. This is something similar to what is achieved with *blending* functions in 3D graphic APIs (like OpenGL).

PDF/A forbids the use of transparency at all. All (semantic) requirements list keys and values that are not allowed to appear in `XObject` or `ExtGState` dictionaries. The value *Transparency* is forbidden for may keys, and possible *alpha* values, which usually indicate a transparency factor in common colour spaces, are forced to have a value of 1.0.

Transparency may seriously affect further rendering of PDF/A compliant files. However, other statically based techiques, like including pre-rendered data or flattened vector objects, are permitted.

4.4.5 Annotations

An *annotation* associates an object such as a note, sound, or movie with a location on a page of a PDF document, or provides a means of interacting with the user via the mouse and keyboard. PDF includes a wide variety of standard annotation types.

Displaying the annotation contents (hold in the `Contents` key of the annotation dictionary) is performed by the reader. PDF 1.4 defines a variety of contents that can be hold in annotations, such text, movies or sounds. PDF/A disallows its usage, because support for multmedia content is out of the scope of the standard.

Additionally, some flags described in PDF 1.4 that manage visibility of annotations are restricted in PDF/A. The intention is to prevent the use of annotations that are hidden or that are viewable but not printable.

4.4.6 Actions

Some PDF objects can specify an *action* for the viewer application to perform, such as launching an application, playing a sound, or changing an annotation's appearance state. The most common usage of actions is jumping to a destination in the document, *e.g.* going to a specific section from the table of contents, centering view in a page footer after clicking a footnote reference, and so on. But actions can also be used, for instance, for launching a web browser and load an URL that the user has selected in the document.

Since multimedia is not supported by PDF/A, the `Launch`, `Sound`, `Movie`, `ResetForm`, `ImportData` and `JavaScript` actions are not permitted. `Launch` allows opening external applications. The `ResetForm` action changes the rendered appearance of a form. The `ImportData` action imports form data from an external file. `JavaScript` actions permit an arbitrary executable code that has the potential to interfere with reliable and predictable rendering.

The only supported named actions are `NextPage`, `PrevPage`, `FirstPage`, and `LastPage`, and the appropriate associated behaviour for each must be performed by the conforming reader.

4.4.7 Metadata

A PDF document may include general information such as the document's title, author, and creation and modification dates. Such global information about the document itself (as opposed to its content or structure) is called *metadata*, and is intended to assist in cataloguing and searching for documents in external databases. A document's metadata may also be added or changed by users or plug-in extensions.

Metadata is essential for effective management of a file throughout its life cycle. A file depends on metadata for identification and description, as well as for describing appropriate technical and administrative matters. As a result, writers of conforming files may have to comply with various domain-specific metadata requirements defined external to `[ISO19005-1:2005]`. These metadata are almost gathered in the XMP specification [59, 3].

4.4.8 Logical structure

ISO 19005-1:2005 establishes two levels of conformance: PDF/A Level A (PDF/A-1a) and PDF/A Level B (PDF/A-1b). PDF/A-1a is even more restrictive than PDF/A-1b. Require-

ments in this subsection are applicable only for files meeting PDF/A Level A conformance. For Level B conformance these requirements can be ignored.

The intent of the requirements in this subsection is to ensure the recovery of the textual content of a conforming file as a sequence of words defined in the natural reading order of the language in which they are written. Similarly, it ensures that the individual characters of each word are recoverable in their natural reading order. Furthermore, these requirements allow the recovery of higher-level semantic information concerning the logical structure of the document.

4.4.9 Interactive Forms

An *interactive form* (sometimes referred to as an AcroForm) is a collection of fields for gathering information interactively from the user. A PDF document may contain any number of fields appearing on any combination of pages, all of which make up a single, global interactive form spanning the entire document. Arbitrary subsets of these fields can be imported or exported from the document.

The intent of the requirements of PDF/A with respect to interactive forms is to ensure that there is no ambiguity about the rendering of form fields. Some related dictionary keys are forbindden or forced to have a *false* value, but most important is the restriction that every form field must have an appearance dictionary associated with the field's data. Then, a conforming reader must render the field according to the appearance dictionary, without regard to the form data. This ensures the reliable rendering of the form.

4.5 A Conformance Module proposal

The main idea behind a formal comparison between all PDF standards and specifications is making the GNU PDF Library completely standard-aware. However, this is a complex goal. On one hand, the file format defined in Adobe documents or in ISO 32000-1:2008 [2] is very extensive. On the other hand, many subsets have been defined in the past years, some are still under development and, for sure, many more will come in the future, as shown in section 4.3. The GNU PDF Library has to provide a set of facilities to application programmers to deal with these standards, subsets and specifications. These facilites must provide support for writing or reading PDF files in a standard-aware manner, so standard requirements have to be constantly checked in any operation that may potentially break the standard conformance.

Morever, it is very desirable to make the GNU PDF Library able to be easily configured by users, enabling and disabling active PDF subsets, standards or specifications. Changing the conformance context, indicating a conformance level to be checked, is a key feature to provide to GNU PDF Library users.

For example, a library user may need to call a function to create a new PDF document. That function should be aware of the base PDF version and subset, in order to generate conformant PDF objects (and therefore a conformant PDF document and file). Application users should be noticed about particular purposes of each standard, allowing them to choose the one that better fits their needs.

The approach that the GNU PDF project is following pretends to build a data structure holding all requirements (restrictions, capabilites) contained in these formal documents, specially specifications and standards. The table contributed to the GNU PDF Knowledge base [5] is the first step towards this data structure. This data structure can be envisaged as a

two-dimensional array M, in which columns represent conformance levels, and rows represent requirements. Given two indexes, i and j ($0 \leq i \leq n-1$ and $0 \leq j \leq m-1$, where n is the number of requirements supported, and m the number of conformance levels supported), then the position $M[i][j]$ holds the status required for the requirement i in the conformance level j (for instance, a *required* value, or an *optional* value). This way, other modules of the library, as well as application programmers, can query the library for the conformance status of any requirement in any supported standard or subset.

The following subsection contains a proposal of a Conformance Module, which should be located in the base layer. Any overlying tier of the library, or any external application built on the top of it, may use its functions or data types to achieve the goals described above. The submitted API is still under reviewing process by the GNU PDF community.

4.5.1 Conformance Module API

The Conformance Module offers operations and data types for configuring the Library in order to ensure conformance with several PDF standards. From a chosen conformance level, requirements that shall, should or shall not be met are given.

The Conformance Module consists of the following parts:

- Requirements Management
- Conformance Context Management
- Conformance-Requirements Mapping Management

Requirements Management

Requirements contained in the standards, ISOs, etc. supported by the Library are parametrized within this module.

Data type: `pdf_req_t`
Opaque type representing a requirement. This data type should contain information needed to uniquely identify a requirement, provide a textual description of the requirement and classify the requirement in a number of categories.

Data type: `pdf_req_repo_t`
Opaque type representing a requirement repository. This data type should contain data representing all the conformance requirements currently set up in the library. Methods for accessing these requirements should be also provided.

Function: `pdf_req_t pdf_req_new (pdf char t * text)`

Create a new requirement with specified parameters.

Parameters

text : Textual description of the requirement.

Returns

The newly created requirement.

Usage example

```
pdf_char_t *desc = "The LZWDecode filter shall not be permitted.";
pdf_req_t req = pdf_req_new (desc);
```

Function: `pdf_req_repo_t pdf_req_repo_new (void)`

Create a new requirement repository.

Returns

The newly created requirement repository.

Usage example

```
pdf_req_repo_t rep = pdf_req_repo_new ();
```

Function: `void pdf_req_add (pdf_req_repo_t rep, pdf_req_t req)`

Add the specified requirement into the given requirements repository.

Parameters

rep : Repository to add the requirement.

req : Requirement to be added.

Returns

Nothing.

Usage example

```
pdf_char_t *desc = "The LZWDecode filter shall not be permitted.";
pdf_req_t req = pdf_req_new (desc);
pdf_req_repo_t rep = pdf_req_repo_new ();

pdf_req_add (rep, req);
```

Function: `void pdf_req_del (pdf_req_repo_t rep, pdf_req_t req)`

Remove the specified requirement from the requirements repository.

Parameters

rep : Repository to delete the requirement.
req : Requirement to be removed.

Returns

Nothing.

Usage example

```
pdf_req_t req;
pdf_req_repo_t rep = pdf_req_repo_new ();

/* ... req is added to rep ... */

pdf_req_del (rep, req);
```

Function: `pdf_char_t * pdf_req_description (pdf_req_t req)`

Obtain the textual description of a given requirement.

Parameters

req : Requirement from which to extract the description.

Returns

A null-terminated char array with the requirement description.

Usage example

```
pdf_req_t req;
printf("Requirement states: %s",
pdf_req_description (req));
```

Conformance Context Management

Conformance of PDF files managed by the Library can be enabled, disabled and configured in order to allow appropriate checks to be performed.

Data type: `enum pdf_cfm_lvl_e`

Enumeration of the supported PDF conformance levels:

`PDF_A_1a` Conformance level PDF/A-1a, as defined by ISO 19005-1:2005.
`PDF_A_1b` Conformance level PDF/A-1b, as defined by ISO 19005-1:2005.
`PDF_1_4` Conformance level PDF 1.4, as defined by PDF Reference, Third Edition, Adobe.

Function: `void pdf_cfm_setcontext (pdf_req_repo_t rep, pdf_cfm_lvl_e level, pdf_bool_t rec)`

Set the conformance context. All operations to follow are affected by the currently set context.

Parameters

rep : A requirement repository.
level : A valid `pdf_cfm_lvl_e` value.
rec : Consider recommendations to be musts.

Returns

Nothing.

Usage example

```
pdf_req_repo_t rep = pdf_req_repo_new ();

/* ... add some requirements to rep ... */

pdf_cfm_setcontext (rep, PDF_A_1a, false);
void pdf_cfm_enable (void);
```

Function: `void pdf_cfm_enable (void)`

Enable conformance checking. This opeartion sets up all requirement values needed for all conformance levels.

Returns

Nothing.

Usage example

```
pdf_cfm_enable ();
```

Function: `void pdf_cfm_disable (void)`

Disable conformance checking.

Returns

Nothing.

Usage example

```
pdf_cfm_disable ();
```

Conformance-Requirements Mapping Management

This submodule allows to perform a mapping between a conformance level and a specified requirement. For a given pair consisting of a conformance level and a requirement, a value is stored indicating if the requirement is a must, a should, etc. Convenience methods for mapping sets of requirements into sets of conformance levels are provided as well.

Data type: `enum pdf_cfm_stat_e`

Enumeration of the supported PDF requirements statuses:

`PDF_CFM_MUST` A requirement that must be satisfied in a given conformance level.

`PDF_CFM_MANDATORY` A requirement that should be satisfied in a given conformance level. If it is not, the PDF file continues to be compliant.

`PDF_CFM_RECOMMENDED` A recommended requirement in a given conformance level. If it is not satisfied, the PDF file is still compliant.

Function: `void pdf_cfm_setvalue_pair (pdf_req_repo_t rep, pdf_cfm_lvl_e level, pdf_req_t req, pdf_cfm_stat_e stat)`

Map the specified conformance level with the supplied requirement with the stat value.

Parameters

rep : A requirement repository.

level : A valid conformance level `pdf_cfm_lvl_e` value.

req : A requirement priviously added to repository.

stat : The status to assign to level req mapping.

Returns

Nothing.

Usage example

```
pdf_req_t req;
pdf_req_repo_t rep = pdf_req_repo_new ();

pdf_req_add (rep, req);
pdf_cfm_setvalue_pair (rep, PDF_A_1a, req, PDF_MUST);
```

Function: `void pdf_cfm_setvalue_level (pdf_req_repo_t rep, pdf_cfm_lvl_e level, pdf_cfm_stat_e * stat)`

Map the specified conformance level with all requirements with the stat array values.

Parameters

rep : A requirement repository.

level : A valid conformance level `pdf_cfm_ lvl_e` value.

stat : An array of statuses to assign to all requirements of *level*.

Returns

Nothing.

Usage example

```
pdf_req_repo_t rep = pdf_req_repo_new ();

pdf_cfm_stat_e * stats = {PDF_CFM_MUST, PDF_CFM_SHOULD, ...};
pdf_cfm_setvalue_level (rep, PDF_A_1a, stats);
```

Function: `void pdf_cfm_setvalue_req (pdf_req_repo_t rep, pdf_req_t req, pdf_cfm_stat_e * stat)`

Map the specified requirement with all conformance levels with the stat value.

Parameters

rep : A requirement repository.
req : A requirement priviously added to repository.
stat : An array of statuses to assign to all conformance levels of *req*.

Returns

Nothing.

Usage example

```
pdf_req_t req;
pdf_req_repo_t rep = pdf_req_repo_new ();
pdf_cfm_stat_e * stats = {PDF_CFM_MUST, PDF_CFM_SHOULD, ...};

pdf_req_add (rep, req);
pdf_cfm_setvalue_req (rep, req, stats);
```

Function: `bool pdf_cfm_check (pdf_req_repo_t rep, pdf_req_t req, pdf_cfm_stat_e stat)`

Check if the specified requirement has a `stat` status in the given requirement repository or not.

Parameters

rep : A requirement repository.
req : A requirement priviously added to `rep`.
stat : A status to check.

Returns

`PDF_TRUE` if `req` has a `stat` value in `repo`; `PDF_FALSE` otherwise.

Usage example

```
pdf_char_t *desc = "The LZWDecode filter shall not be permitted.";
pdf_req_t req = pdf_req_new (desc);
pdf_req_repo_t rep = pdf_req_repo_new ();

/* Add the requirement to the repository */
pdf_req_add (rep, req);

/* Set the status of the requirement in the repo for a given conformance
   level */
pdf_cfm_setvalue_pair (rep, PDF_A_1a, req, PDF_MUST);

/* Check */
if (pdf_cfm_check (rep, req, PDF_MUST)
{
    /* Compress stream without LZW. */
}
```

Chapter 5

A PDF object parser proposal

A key feature of any library that manages a file format is to provide mechanisms to read files in that format and process its contents. The most common example is a programming language source code file: the tools that confom the compiler read an input stream, extract tokens from it, perform parsing and check for possible compiling-time semantic errors. At the beginning of this collaboration FDP, the GNU PDF Library did not have such a format-validation process, though some tools –like filtered streams and a tokeniser– were already working.

This chapter describes the process of building a PDF object parser for the GNU PDF Library.

5.1 Introduction

A file format is a particular way that information is encoded for storage in a computer file [60]. Moreover, it is a set of lexical, syntactic and semantic rules that must be met by the data contained in that file.

Checking the correctness of data with respect to a given format can be a complex process. Typically, computer programs are written in source code files in some programming language (*e.g.* the GNU PDF Library is written in the C language). This source code must be read and validated before doing specific processing. For instance, after lexical, syntactical and semantic checks, a programmer that has written a program in the C language may want the compiler to generate intermediate code in order to get an executable file; otherwise, a Java programmer may want to generate Virtual Machine (VM) bytecode. These are the typical phases needed from a compiler, shown at figure 5.1 (extracted from [4]).

However, file formats designed to contain *data* (instead of program instructions) rarely need phases like code generation. Since the goal in these file formats is to appropriately represent some kind of data (images –JPEG or PNG–, hyerarchical structures –XML–, webpage content –HTML–, or document containers –PDF–), after the lexical, syntactic and semantic stages follows a stage were data structures neeeded to represent these data are generated. Then, reader and writer programs use these data structures to represent the contents of files.

The PDF file format is well defined with respect to lexical components [2]. Moreover, at the beginning of this collaboration the GNU PDF Library was already providing a fully functional tokeniser module. This module, according to figure 5.1, is capable of providing tokens, as the input stream is being read. Hence, the lexical analysis was being correctly

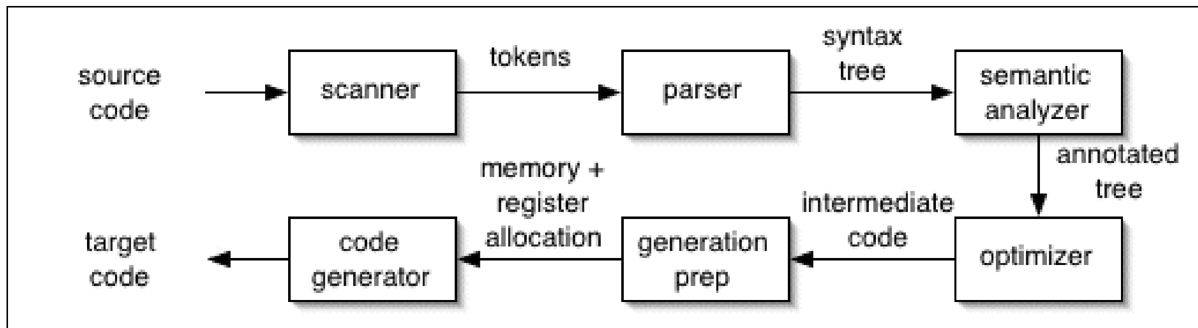


Figure 5.1: Phases of a compiler.

performed before this FDP entered the scene.

The next logical step was, then, the syntactic analysis. Developing a parser made sense at this point, according with figure 5.1. Additionally, many users (not necessarily contributors) were entering the `pdf-devel` mailing list asking if the GNU PDF Library was capable of PDF parsing or not. Unfortunatelly, it was not.

A proposal for developing a PDF object parser was agreed with the GNU PDF community. It is important to note that the library had (and still has) many parsing needs, not only related with PDF objects parsing. The following list of needed parsers tries to clarify this, while it helps to define too what a PDF object parser exactly is.

- There was a need for a **PDF file parser**. PDF files, as explained in chapter 4, consist essentially of a *header*, a *body* (which contains all PDF objects conforming the document), and a *cross-reference table*. Parsing a PDF file means, according with this, to parse the header, parse the body, and finally parse the cross-reference table, though not necessarily in that order. A sequential parsing of a PDF file is not the desired way to proceed by the GNU PDF community, though many PDF libraries read PDF files following a similar procedure. One of the main benefits of the PDF file format is that any part of the document contained in a PDF file can be rendered without having to load the entire file content. This is achieved combining the information of the cross-reference table and the body: the former contains references (offsets) to quickly access any object in the body (remember that the body is simply a collection of PDF objects). All information needed to render a page is contained in a reduced set of PDF objects of the body, quickly accessible through the offsets of the cross-reference table. This suggests that the best parsing strategy implies parsing the cross-reference table *first*. Then, PDF objects are parsed *later*, just when the reader needs them (not all of them; only those needed to render requested pages). Parsing the file header is a trivial task, out of the interest of this FDP. Additional information on basic PDF file reading can be found at the GNU PDF Knowledge base [36].

- There was a need for a **PDF object parser**. Once the reader knows which PDF objects are needed to render the current page or the document section requested, those objects must be accessed and loaded. Moreover, syntax checking of PDF objects must be done at this point, since invalid formatting may appear within them. The PDF object parser must be capable to receive a piece of stream containing the PDF object. More precisely, it receives the offset of that object specified in the cross-reference table, and tries to process it, token by token. If these tokens meet the rules of a valid PDF object

grammar, then the syntactic check passes for that PDF object, and an appropriate data structure representing that object is returned. Otherwise, a syntactic error is returned, and some error recovery procedure is used to generate a valid data structure (the PDF object that corresponds to that structure continues to be invalid). If the syntactic error is not recoverable, an error is returned and the PDF object is not processed at all (*i.e.* no data structure is generated for that object).

- There was a need for a **PDF content streams parser**. As explained below (see section 5.2, there is one type of PDF objects that deserve special attention: *stream objects*. Stream objects are basically sequences of bytes, and thus have the peculiarity of store almost any kind of data. This includes very heterogeneus contents, from JPEG images to PostScript fragments, JavaScript code, video, sound, XML metadata, and so on. Once decoded, these contents may need specific parsing routines (*i.e.* JavaScript code has to be validated by a JavaScript parser). Hence, the PDF content streams parser is, in fact, a set of parsers making syntax checking for the vast typology of PDF embedded media. Fortunately, most of these parsers already exist as free software, so they can be linked when it becomes necessary.

These parsing activities were prioritized, in order to clarify its importance and urgency. Moreover, some of these tasks have dependencies with other parts of the library that were not developed enough when the collaboration began, and had to be consequently postponed. The following list shows suitability issues for each needed parser.

- Regarding the **PDF file parser**, two issues were found. First, and most important, the GNU PDF Library was not mature enough to develop the functionality of parsing an entire PDF file. Moreover, providing cross-reference parsing could make the programmers think erronously that the library was at a development stage advanced enough to parse PDF files. Second, offering PDF file parsing did not make sense since the base layer was still focusing almost all development efforts, with some contributors beginning to work in the object layer.

- Some pros and cons were found considering the **PDF object parser** developing. First, it is a dependency to be fixed before developing a PDF file parser, since the PDF file parser needs the PDF object parser to work properly (remember that parsing a PDF file involves parsing PDF objects, the file header and the cross-reference table, though not necessarily in that order). Second, it offered enough monolithic developing, from the point of view that it does not require linking many external libraries. Third, a considerable part of the work developed in chapter 4, specifically syntax rules in ISO 32000-1:2008 [2], could be very useful in a PDF object grammar definition. As the only con, the PDF object parser relies on PDF content streams parsers, that had to be linked in a next stage.

- Regarding the **PDF content streams parser**, two major issues were found. First, most of the work needed to encode or decode stream contents was already implemented in the base layer. Second, syntax checking of decoded stream contents could be as easy as invoking specific,exteranl parsers from free libraries. This made this option less attractive than the PDF object parser proposal.

According with all this, a **PDF object parser** was found to have major priority among other parsing needs. Additionally, it may use many components developed in the base layer specifically for PDF object parsing purposes (*i.e.* the tokeniser), which is an insteresting issue to both understand and test library modules. Once implemented and tested, PDF file parsing and stream content parsing may be easily developed and integrated as next steps, as the library continues to rise in its stack, bottom-up schema.

A *formal grammar* specifying the rules defining the PDF object language is necessary before developing a PDF object parser. Moreover, the formal grammar design stage has a critical influence on the parser itself. On one hand, the grammar nature may decide the best parsing strategy, and on the other hand, choosing a concrete parsing algorithm may require modifications on the grammar to get it correctly adapted to algorithm's needs [4, 6].

The term *PDF language* is going to be used in this chapter as a way to summarize *the set of all input strings that are valid with respect to official PDF specifications*. That is, from a formal language point of view, the language accepted by a PDF grammar that contains rules defining the PDF objects format.

The chapter is organized as follows. First, a summary of the PDF objects that can be found in any PDF file is presented. Second, the work on developing a grammar that accepts the valid PDF language described in official specifications, and only such a language, is described. Finally, the PDF object parser program and all neede stages for its development are shown.

5.2 PDF objects

Previously, it has been shown that PDF files contain a *body* section. This body consists of a set of PDF objects, which abstractly represent all elements that conform the PDF document. This section enumerates and describes all kinds of PDF objects defined by PDF standards. To understand better how these objects are accessed to form the PDF document, please see the GNU PDF Knowledge base introduction about PDF files [36].

PDF includes eight basic types of objects: Boolean values, Integer and Real numbers, Strings, Names, Arrays, Dictionaries, Streams, and the null object.

Objects may be labelled so that they can be referred to by other objects. A labelled object is called an indirect object (see 5.2.9).

5.2.1 Boolean objects

Boolean objects represent the logical values of true and false. They appear in PDF files using the keywords `true` and `false`.

5.2.2 Numeric objects

PDF provides two types of numeric objects: integer and real. *Integer objects* represent mathematical integers. *Real objects* represent mathematical real numbers. The range and precision of numbers may be limited by the internal representations used in the computer on which the conforming reader is running.

An integer shall be written as one or more decimal digits optionally preceded by a sign. The value shall be interpreted as a signed decimal integer and shall be converted to an integer object.

A real value shall be written as one or more decimal digits with an optional sign and a leading, trailing, or embedded decimal point. The value shall be interpreted as a real number and shall be converted to a real object.

Examples

```
123 43445 +17 -98 0

34.5 -3.62 +123.6 4. -.002 0.0
```

5.2.3 String objects

A *string object* consists of a series of zero or more bytes. String objects are not integer objects, but are stored in a more compact format.

String objects shall be written in one of the following two ways:

- As a sequence of literal characters enclosed in parentheses `( )`. These are called *Literal Strings.*
- As hexadecimal data enclosed in angle brackets `< >`. These are called *Hexadecimal Strings.*

Examples

```
( Strings may contain balanced parentheses ( ) and
special characters ( * ! & } ^ % and so on ) . )

< 4E6F762073686D6F7A206B6120706F702E >
```

5.2.4 Name objects

A *name object* is an atomic symbol uniquely defined by a sequence of any characters (8-bit values) except null (character code 0). Uniquely defined means that any two name objects made up of the same sequence of characters denote the same object. Atomic means that a name has no internal structure; although it is defined by a sequence of characters, those characters are not considered elements of the name.

When writing a name in a PDF file, a slash (/) shall be used to introduce a name. The slash is not part of the name but is a prefix indicating that what follows is a sequence of characters representing the name in the PDF file.

Examples

```
/Name1

/A;Name_With-Various***Characters?
```

5.2.5 Array objects

An *array object* is a one-dimensional collection of objects arranged sequentially. Unlike arrays in many other computer languages, PDF arrays may be heterogeneous; that is, elements of an array may be any combination of numbers, strings, dictionaries, or any other objects, including other arrays. An array may have zero elements.

An array shall be written as a sequence of objects enclosed in square brackets (`[ ]`).

Examples

```
[ 549 3.14 false ( Ralph ) /SomeName ]
```

5.2.6 Dictionary objects

A *dictionary object* is an associative table containing pairs of objects, known as the dictionary's *entries*. The first element of each entry is the *key* and the second element is the *value*. The key shall be a name (unlike dictionary keys in PostScript, which may be objects of any type). The value may be any kind of object, including another dictionary. A dictionary entry whose value is `null` shall be treated the same as if the entry does not exist.

The entries in a dictionary represent an associative table and as such shall be unordered even though an arbitrary order may be imposed upon them when written in a file. That ordering shall be ignored.

Multiple entries in the same dictionary shall not have the same key. A dictionary shall be written as a sequence of key-value pairs enclosed in double angle brackets (`<< ... >>`).

Examples

```
<< /Type /Example
   /Subtype /DictionaryExample
   /Version 0.01
   /IntegerItem 12
   /StringItem ( a string )
   /Subdictionary << /Item1 0.4
                     /Item2 true
                     /LastItem ( not ! )
                     /VeryLastItem ( OK )
                  >>
>>
```

5.2.7 Stream objects

A *stream object*, like a string object, is a sequence of bytes. Furthermore, a stream may be of unlimited length, whereas a string shall be subject to an implementation limit. For this reason, objects with potentially large amounts of data, such as images and page descriptions, shall be represented as streams.

A stream consists of a dictionary followed by zero or more bytes bracketed between the keywords `stream` (followed by newline) and `endstream`:

dictionary

stream
... Zero or more bytes ...
endstream

All streams shall be indirect objects (see 5.2.9) and the stream dictionary shall be a direct object. The keyword `stream` that follows the stream dictionary shall be followed by an end-of-line marker. The sequence of bytes that make up a stream lie between the end-of-line marker following the stream keyword and the endstream keyword; the stream dictionary specifies the exact number of bytes. There should be an end-of-line marker after the data and before endstream; this marker shall not be included in the stream length. There shall not be any extra bytes, other than white space, between `endstream` and `endobj`.

Alternatively, beginning with PDF 1.2, the bytes may be contained in an external file, in which case the stream dictionary specifies the file, and any bytes between `stream` and `endstream` shall be ignored by a conforming reader.

Examples

```
<<
  /Length 44
>>
stream
BT
70 50 TD
/F1 12 Tf
(Hello, world!) Tj
ET
endstream
endobj
```

5.2.8 Null object

The *null object* has a type and value that are unequal to those of any other object. There shall be only one object of type null, denoted by the keyword `null`. An indirect object reference (see 5.2.9) to a nonexistent object shall be treated the same as a null object. Specifying the null object as the value of a dictionary entry (see 5.2.6) shall be equivalent to omitting the entry entirely.

5.2.9 Indirect objects

Any object in a PDF file may be labelled as an *indirect object*. This gives the object a unique object identifier by which other objects can refer to it (for example, as an element of an array or as the value of a dictionary entry).

The object identifier shall consist of two parts:

- A positive integer *object number*. Indirect objects may be numbered sequentially within a PDF file, but this is not required; object numbers may be assigned in any arbitrary order.

- A non-negative integer *generation number*. In a newly created file, all indirect objects shall have generation numbers of 0. Nonzero generation numbers may be introduced when the file is later updated.

Together, the combination of an object number and a generation number shall uniquely identify an indirect object.

The definition of an indirect object in a PDF file shall consist of its object number and generation number (separated by white space), followed by the value of the object bracketed between the keywords `obj` and `endobj`.

The object may be referred to from elsewhere in the file by an *indirect reference*. Such indirect references shall consist of the object number, the generation number, and the keyword `R` (with white space separating each part):

```
12 0 R
```

Examples Indirect object definition

```
12 0 obj
     ( Brillig )
endobj
```

Defines an indirect string object with an object number of 12, a generation number of 0, and the value Brillig.

Indirect reference to this indirect object

```
12 0 R
```

5.3 A grammar for PDF objects

The PDF objects grammar proposed in this section accepts an input string S if and only if S is a valid PDF object. Otherwise, the grammar does not accept S.

At this stage, the input string has been processed by the tokeniser module in the base layer, performing the lexical analysis and splitting the input stream into tokens. Hence, the input string S for the PDF object grammar consists of a sequence of tokens. All possible tokens collected by the tokeniser are terminal symbols for the object grammar. Therefore, let Σ be the following terminal symbols alphabet:

$$\Sigma = \{TRUE, FALSE, STREAM, ENDSTREAM, NULL, OBJ, ENDOBJ, \\ R, INTEGER, REAL, STRING, NAME, DICT_START, \\ DICT_END, ARRAY_START, ARRAY_END, \lambda\}$$

Table 5.1 shows the correspondence between the tokens (terminal symbols) of Σ and the chunks of the input string S read by the tokeniser.

This token set is not arbitrary. Moreover, the GNU PDF Library had it defined and processed by the tokeniser module before this collaboration began. The design criteria of this lexical processing was:

- All token instances (`pdf_token_t` data type) hold a *value* (*i.e.* `-3.14` for an integer object) and a *type* (*i.e.* the `integer` type) for each token recognized in the input stream.

Token	Input string chunk
TRUE	`true`
FALSE	`false`
STREAM	`stream`
ENDSTREAM	`endstream`
NULL	`null`
OBJ	`obj`
ENDOBJ	`endobj`
R	`R`
INTEGER	*An integer number*
REAL	*A real number*
STRING	*A literal or hexadecimal string*
NAME	*A name object*
DICT_START	`<<`
DICT_END	`>>`
ARRAY_START	`[`
ARRAY_END	`]`
λ	

Table 5.1: Correspondence between tokens of Σ and input string S chunks.

- Token types for **integers**, **reals**, **strings** (with no explicit distinction between literal nor hexadecimal) and **names** correspond with their equivalent PDF object classes.
- There is a special type of tokens, **keywords**, that is created with the tokeniser recognizes the keywords `true`, `false`, `stream`, `endstream`, `null`, `obj`, `endobj` and `R`. The information about what specific keyword was recognized can be queried in the token value (which holds the keyword value read from the input stream).
- Arrays and dictionaries require syntactic analysis to be accepted or refused, so the only tokens recognized in the lexical stage are the dictionary and array start and end delimiters (`<<`, `>>`, `[`, `]`).
- Other token types recognized by the tokeniser, such as comments and procedure start and end delimiters, are omitted in PDF object parsing.

Two proposals for a PDF object grammar follow. A summarized version of these grammars can be found in appendix D. Let G_1 and G_2 be BNF (Backus-Naur Form) grammars for the first and second proposals, respectivelly.

5.3.1 First proposal, G_1

On one hand, the first proposal makes G_1 more readable and understandable; on the other hand, G_1 rules are unrolled and over redundant. Second proposal in subsection 5.3.2 shows more compacted rules in G_2.

The first rule of G_1 defines and *indirect object*. Indirect objects are the form in which objects are found in a PDF file. This is because a conforming reader only can access objects defined in the cross-reference table, and those must be indirect objects. As stated in [2], *the*

definition of an indirect object in a PDF file shall consist of its object number and generation number (separated by white space), followed by the value of the object bracketed between the keywords **obj** *and* **endobj**. This leads to the rule defined in figure 5.2.

indirect_object → *INTEGER INTEGER OBJ* (
(*TRUE* | *FALSE*)
| *INTEGER* (*INTEGER R* | λ)
| *REAL*
| *STRING*
| *NAME*
| *NULL*
| *array_object*
| *dictionary_object* (*stream_object* | λ)
) *ENDOBJ*

Figure 5.2: Indirect object rule for G_1.

Object and generation numbers are given by the first two integers. Following them, the keyword **obj** represented by the *OBJ* token introduces the value of the object itself. As the rule states, this object value shall be either a *boolean object* (*TRUE* and *FALSE* tokens), an *integer object* (*INTEGER* token), an *indirect reference* (*INTEGER*, *INTEGER* and *R* tokens), a *real object* (*REAL* token), a *string object* (*STRING* token), a *name object* (*NAME* token, a *null object* (*NULL* token), an *array object* (*array_object* rule), a *dictionary object* (*diciontary_object* rule), or a *stream object* (*dictionary_object* and *stream_object* rules). Finally, a mandatory **endobj** keyword, represented with the *ENDOBJ* token, closes the indirect object.

Indirect objects are the top level objects in the object grammar G_1.

The second rule of G_1 defines an *array object*. An array object is a one-dimensional collection of objects arranged sequentially. Unlike arrays in many other computer languages, PDF arrays may be heterogeneous; that is, an array's elements may be any combination of numbers, strings, dictionaries, or any other objects, including other arrays. *An array may have zero elements. An array shall be written as a sequence of objects enclosed in square brackets*[2]. This defines array objects as shown in figure 5.3.

array_object → *ARRAY_START* (*array_or_dictionary_value*) ∗ *ARRAY_END*

Figure 5.3: Array object rule for G_1.

The left square bracket (`[`) represented with the *ARRAY_START* token marks the beginning of the array object. It may contain a list of zero or more *array_or_dictionary_value*. Finally, a right square bracket (`]`) (*ARRAY_END* token) closes the array object. Candidate values that may be enclosed in an array object (also in a dictionary object, as shown below) are generated through the third rule of G_1 (see figure 5.4.

One should note here that an *array_or_dictionary_value* and the possible contents of an *indirect_object* are almost the same, though not exactly equal. In fact, the only difference between candidate objects enveloped inside an *indirect object* (see figure 5.2) and objects generated by the rule *array_or_dictionary_value* (see figure 5.4) is the suffix (*stream_object* | λ) that can optionally appear in the former.

Some important design details should be noted here:

array_or_dictionary_value $\rightarrow$ ($TRUE$ | $FALSE$)
| $INTEGER$ ($INTEGER$ R | λ)
| $REAL$
| $STRING$
| $NAME$
| $NULL$
| *array_object*
| *dictionary_object*

Figure 5.4: Array / dictionary value rule for G_1.

1. The common options in both rules describe a set of simple, atomic objects (*boolean objects*, *integer objects* and *real objects*, *indirect references*, *string objects*, *name objects* and *name objects*), and two one-dimensional, collection data types (*array objects* and *dictionary objects*). One may think about an independent rule gathering these types. However, some problems arise with such a rule, mainly related with ambiguities. As this can be seen as an improvement to G_1, this issue is discussed in section 5.3.2.

2. *Stream objects* are always indirect objects, and thus they can not be grouped together with the rest of (non-indirect) objects. Moreover, one must be sure that a call to rule *stream_object* must be performed always after a call to *indirect_object*. Equivalently, this is the same than assuring that any *stream_object* node in any derivation tree must be a child (or contained in a subtree) of an *indirect_object* node.

3. Some ambiguities arise in the grammar when trying to define a *stream_object* in a rule as a *dictionary_object* plus some other tokens, as doing this leads to several rules beginning with the $DICT_START$ token.

4. Using a single rule grouping all kinds of objects pointed in the first rule of G_1 (see figure 5.2), and using this rule to define objects that can be contained in *array objects* or *diciontary objects* leads to invalid derivation trees, as *stream objects* are not allowed in these data types.

The fourth rule of G_1 defines the syntax of *dictionary objects* (see figure 5.5). A *dictionary object* is an associative table containing pairs of objects, known as the dictionary's *entries*. The first element of each entry is the *key* and the second element is the *value*. The key shall be a *name object*. The value may be any kind of object, including another dictionary. A dictionary may have zero entries. A dictionary shall be written as a sequence of key-value pairs enclosed in double angle brackets (`<< ... >>`) [2].

dictionary_object $\rightarrow$ $DICT_START$ (*key_value_pair*) $*$ $DICT_END$

Figure 5.5: Dictionary object rule for G_1.

The rule for a *dictionary object* describes the contents of the dictionary as a list of zero or more elements provided by the *key_value_pair* rule, which is shown in figure 5.6.

A *key_value_pair* is defined as a *name object* followed by an *array_or_dictionary_value*, previously described in figure 5.4.

Finally, a *stream object* must be strictly called after a *dictionary object* [2]; this is ensured by the first rule of G_1 (see figure 5.2). The contents of a *stream object* are not parsed in

key_value_pair → *NAME array_or_dictionary_value*

Figure 5.6: Key-value pair rule for G_1.

the PDF object grammar. Content streams are usually filtered sequences of bytes that need a special treatment; they are parsed later in the GNU PDF Library, more precisely in the *document* and *page* layers of the library. They are pointed here in a symbolic way by the *content_stream* rule (see figure 5.7).

stream_object → *STREAM* (*content_stream*) ∗ *ENDSTREAM*

Figure 5.7: Stream object rule for G_1.

5.3.2 Second proposal, G_2

The first rule of G_2 also defines an *indirect object*, the top-level object type found in a PDF file body. However, types of objects that can be contained in an indirect object are specified by means of a new rule that was not present in G_1, named *contained_object.* This first rule is shown in figure 5.8.

indirect_object → *INTEGER INTEGER OBJ* (*contained_object*)*ENDOBJ*

Figure 5.8: Indirect object rule for G_2.

Again, object and generation numbers are given by the first two integers. Following them, the keyword **obj** introduces the value of the object itself. However, this time a variable *contained_object* is found in this place. *Contained objects* is a class of PDF objects that does not belong to any official PDF specification; but defining it helps with dealing with other types of objects and resolving ambiguities. A *contained object* (that is, an object that can be contained between indirect object keyword delimiters) is defined as shown in figure 5.9.

contained_object → *atomic_object*
| *array_object*
| *dictionary_object* (*stream_object* | λ)

Figure 5.9: Contained object rule for G_2.

Hence, a *contained object* may be either:

- An *atomic object*, which is another dummy class for atomic objects,
- or an *array object*,
- or a *dictionary object*,
- or a construction which consists of a *dictionary object* and a *stream object.* This is the syntax for *stream objects*, that shall be *indirect objects* with no exceptions [2] (this means that a *stream object* appears as a *contained object* and nowhere else).

The rule for *atomic objects* is shown in figure 5.10, while the rules for *array objects*, *dictionary objects* and *stream objects* are shown in figures 5.11, 5.13 and 5.15 respectivelly.

atomic_object →
(*TRUE* | *FALSE*)
| *INTEGER* (*INTEGER R* | λ)
| *REAL*
| *STRING*
| *NAME*
| *NULL*

Figure 5.10: Atomic object rule for G_2.

Atomic objects are essentially non-reductible objects (tokens) of classes *boolean object*, *integer object*, *indirect reference*, *real object*, *string object*, *name object* and *null object*. The reason for excluding *array objects* and *dictionary objects* of the *atomic objects* class is discussed later.

array_object → *ARRAY_START* (*array_or_dictionary_value*) * *ARRAY_END*

Figure 5.11: Array object rule for G_2.

An array encloses a list of zero or more *array_or_dictionary_value*. The rule presenting these values is shown in figure 5.12.

array_or_dictionary_value →
atomic_object
| *array_object*
| *dictionary_object*

Figure 5.12: Array or dictionary value rule for G_2.

At this point, the reasons for grouping *atomic objects* can be discussed. Observing rules defining *array objects*, *dictionary objects* and *indirect objects*, one may realize that the types of objects that can be contained within them are almost the same. In a more deep analysis, objects that can be contained within *array objects* and objects that may conform *values* of a *key-value pair* in *dictionary objects* are exactly the same: *boolean objects*, *integer objects*, *real objects*, *indirect references*, *string objects*, *name objects*, *null objects*, *array objects* and *dictionary objects*. For this reason a rule defining *array_or_dictionary_object* is used.

However, objects to be contained inside *indirect objects* admit more options, because the acceptance of *stream objects*, which can only be invoked here. Hence, it is true that *indirect objects* admit all atomic objects plus *array objects* and *dictionary objects*, just like the *array_or_dictionary_object* rule does. But, additionally, *indirect objects* may contain, after the *dictionary object*, a *stream object* construction.

This makes impossible to define a single class of objects containing *atomic objects* plus *dictionary objects*. By doing it, many ambiguities arise: for example, two options beginning with a *dictionary object* from two different rules may apply: one defining this *atomic objects* plus *dictionary objects* hypothetical class, and one defining a *dictionary object* followed by a *stream object* (in the *indirect objects* rule). This is the reason for excluding *dictionary objects* from the *atomic objects* set. In fact, *atomic objects* contain the minimum set of objects that may be invoked as values of *array objects*, *dictionary objects*, and *indirect objects*.

The rest of rules completing G_2 are analogous to the rules defined in G_1 for *dictionary*

objects (see figure 5.13), *key-value pairs* (see figure 5.14), and *stream objects* (see figure 5.15).

dictionary_object → *DICT_START* (*key_value_pair*) ∗ *DICT_END*

Figure 5.13: Dictionary object rule for G_2.

key_value_pair → *NAME array_or_dictionary_value*

Figure 5.14: Key-value pair rule for G_2.

stream_object → *STREAM* (*content_stream*) ∗ *ENDSTREAM*

Figure 5.15: Stream object rule for G_2.

5.3.3 G_1 and G_2 pros and cons

Proposed PDF object grammars G_1 and G_2 can be compared in the following facets:

- **Grouping of options, simplification and redundancy**. It is clear to the reader that grouping of options is more acurately performed in G_2 than in G_1. Actually, the number of rules of G_2 is greater than the number of rules of G_1, which shows that more steps of rule grouping have been done designing G_2. That makes G_2 also more *readable*, but slightly more complex to compute than G_1: derivation trees of G_2 have an average height greater than derivation trees of G_1 for the same input string S. Considering this, it can be stated that G_1 is *simpler* than G_2. However, G_1 is also more redundant, since the same tokens, token groups and choices are repeated over and over again through the grammar.
- **Number of rules**. As a consequence of the previous point, the number of rules of G_2 (8) is greater than the number of rules of G_1 (6). This slightly lows the performance of computing G_2 against G_1, but increases readability of G_2 against G_1.
- **Ambiguity**. Neither G_1 nor G_2 contain ambiguities. To check this, a prototypical implementation was performed using the ANTLR Parser generator [43]. Output is shown in appendix D.
- **Equivalency**. Both grammars G_1 and G_2 are equivallent, since they recognize exactly the same language. Some Abstract Syntax Trees (AST) were generated by G_1 and G_2 prototypical parsers and were identical.

Regarding the **number of rules**, some additional and important remarks make sense with respect to two key issues of a parser:

- **Error recovery**. Depending on the context, when a syntax error occurs some recovervy strategies can be followed. For instance, some tokens can be skipped and thrown away (this implies some data loss, which could affect document structure or rendering), or some may be created artificially, in order to get a consistent and coherent parsing result. Sometimes this is not possible; the parsing process has to be stopped then, and a report must be warn the user about what went wrong.

- **AST construction**. The parsing output is an Abstract Syntax Tree, which specifies, in a binary tree structure, the hierarchical representation of PDF objects parsed.

Increasing the number of rules or, more precisely, establishing rules in a logical and consistent way, provides best results with both error recovery and AST construction. This is specially true for specific parsing strategies, like LL parsing. As it will be shown later (see section 5.4), the selected parsing strategy for the PDF object grammar is easily improved when a higher number of grammar rules help to better read and understand the language that it is defining. Rules help to logically find optimal code locations to insert *error recovery calls* and *AST construction calls*.

These arguments lead to choose G_2 as the PDF object grammar to be adapted and used for the development of a PDF object parser.

5.4 Parser development

When developing parsers for programming languages, it is very common to use some parser-generation tool that reads the language grammar (conveniently adapted for the case) and generates the parser code automatically. However, the GNU PDF Library PDF object parser is written from scratch, instead of using parser generators like GNU Bison [17], Yacc [40], PCCTS [44] or ANTLR [43] (though this last was used for grammar design). The development community took this decision after considering the following:

- Parser code should be optimal regarding resource consumption. This could not be guaranteed using parser generators, which in order to be generic enough generate vast source code files that can be reduced in size and memory usage.
- Parser code should be readable and easily improvable. Automatically generated code does not allow this. On one hand, the way to modify the resulting source code is to change the input file for the parser generator; this does not always provide the desired options. On the other hand, if developers try to modify the automatically generated source code, then a lot of adaptation work may have to be performed. These efforts could be reduced if applied to a from-scratch approach.
- Parser code may be spread across the GNU PDF Library. This means that the parsing tasks are not performed in a single step: as shown above, PDF object parsing goes in some manner after PDF file parsing, and before streams content parsing.

These requirements depict parsing needs which have many peculiarities with respect to *simple* or *standard* programming languages parsing. It also has to be noted that parser generators are designed having in mind programming languages as their primary target. PDF, as shown above, differs in many ways from a usual programming language, and thus it makes sense to find some mismatches with parser generators.

This analysis lead to choose a from-scratch developing for the PDF object parser.

5.4.1 Lookahead

Lookahead establishes the maximum incoming tokens that a parser can use to decide which rule it should use [62]. The term *lookahead* is abbreviated here as *LA*.

Normally, languages are developed trying to avoid situtations where more than one token is needed to determine the next grammar rule to apply. The PDF language conforms with this, excepting one case alone. In the case that follows, ambiguity regarding rule application occurs if there is only one token of lookahead (that is, the current token being processed, which is noted $LA(0)$ here) available.

The ambiguity occurs while processing a sequence of integers, mainly during the validation of the contents of an **array object**. Suppose the following PDF array object, which can be easily found in PDF files as a value of a key-value pair in a dictionary, for instance:

```
[ 2 0 4 5 0 R 9 ]
```

The correct output that the parser should return here is $ARRAY_START$, $INTEGER$, $INTEGER$, $INTEGER$; then a match for the rule defining *indirect references*, which parses the inseparable tokens $INTEGER$, $INTEGER$ and the keyword R; then a final $INTEGER$, and finally the $ARRAY_END$ token.

The problem here is that, with $LA(0)$ alone, it is not possible to determine when to apply the *integer object* matching rule, or the *indirect reference* matching rule, during the processing of the array contents. Moreover, as both rules begin with an $INTEGER$ token, the parser has to choose the first specified (which has more priority). If the first rule in the array contents definition is the one that matches *integer objects* (see 5.3.2), then the parser will process, following always this rule, the integer tokens `2`, `0`, `4`, `5` and `0` before returning an error when receiving the `R` keyword, because does not exist any rule beginning with the `R` keyword in the array contents definition. Moreover, there is no construct in PDF language beginning with that keyword. Otherwise, if the first rule in the array contents definition is the one that matches *indirect references* (see 5.3.2), the integer tokens `2` and `0` are processed before returning an error when receiving the third integer, because the `R` keyword was expected to match the *indirect reference* rule.

With this scenario, it is clear that it is not possible to determine the choice between *indirect reference* or the *integer object* rules with a single $LA(0)$ token for lookahead. Moreover, when the parser reads the integer `5` it has to know that the rule to process it, just at this point, is the *indirect reference* one instead of the *integer object* one. Note that adding one more token to the lookahead, that is, the $LA(1)$ token, is not enough: at this point the parser would see that $LA(0) = 5$ and $LA(1) = 0$, which is clearly insufficient to know if these are either independent integers or the beginning part of an indirect reference. It is clear that, in order to make the parser know that these tokens are, in fact, the beginning part of an indirect reference, an additional token in the lookahead is needed.

Hence, a *parse window* of three tokens is defined. With such a window, the parser can correctly choose the appropriate rule (see figure 5.2). The remaining grammar rules are decidable using $LA(0)$ only in any given PDF context.

Additionally, the implementation of *libpoppler* [39], which features a PDF parser, was consulted regarding the lookahead issue. It was verified that libpoppler's parser also uses two additional tokens for its lookahead.

5.4.2 Parsing strategy

Once a grammar for PDF objects has been specified, and the requirement of writing a parser for it from scratch has been stated, a parsing algorithm can be written. However, there is a number of distinct parsing strategies, each of them with its pros and cons.

Step	$LA(0)$	$LA(1)$	$LA(2)$	**Rule**	**Description**
1	2	0	4	*Integer object*, because the window only contains integers	The parser matches one integer token with value 2. The window advances.
2	0	4	5	*Integer object*, because the window only contains integers	The parser matches one integer token with value 0. The window advances.
3	4	5	0	*Integer object*, because the window only contains integers	The parser matches one integer token with value 4. The window advances.
4	5	0	R	*Indirect reference*, because there are integers in $LA(0)$ and $LA(1)$, and the `R` keyword in $LA(2)$	The parser matches the first integer, it advances, it matches the second, it advances again, it matches the keyword, and advances one more time.
5	9	...	...	*Integer object*, because the window contains and integer and the following tokens ending the array	The parser matches one integer token with value 9. The window advances.

Table 5.2: Parsing window example. The parser uses three tokens as a lookahead to decide which grammar rule to apply.

Generic parsing algorithm

The generic parsing algorithm, also known as the *Cocke-Younger-Kasami* (CYK) algorithm, runs in $O(n^3)$ time, where n is the length of the input string S [6, 4]. As it is known that there exist algorithms with better performance, the use of the CYK schema for the PDF object parser was discarded.

Bottom-up strategies

The most common schema for bottom-up parsing is *LR(1)* parsing. As a bottom-up strategy, it builds the AST from the leaves to the root. It also follows a right-most derivation in backward direction. This means that terminal symbols are converted to non-terminal (variables) symbols, using the rules of the grammar, and grouping from right to left when more than one choice is possible. This process is applied repeatedly to the S input string until the starting symbol of the grammar is derived from it (otherwise, the syntax checking fails).

LR(1) parsing runs in $O(n)$ time, where n is the length of the input string S. Though it is slightly more powerful than *LL(1)* parsing [6] (see below), it is much less intuitive. Moreover, it would require to use some LR parser-generator tool (*e.g.* GNU Bison or Yacc), since *LR* strategy can not be easily written from scratch to adapt itself to a specific language. As this does not fit with GNU PDF Library requirements, the use of *LR(1)* parsing was also discarded.

Top-down strategies

This approach consists of basically two strategies: *table-driven parsing* and *recursive descent predictive parsing*. As the former can be understood, more or less, as an iterative version of the latter, the recursive approach is going to be used here for clarity reasons.

Recursive descent predictive parsing consists, mainly, of *LL(1)* parsing. These parsers build the AST from the root to the leaves (top-down), and follow a left-most derivation in forward direction. This means that, beginning from the starting symbol of the grammar, rules are repeatedly applied until the input string S is matched, replacing non-terminal symbols from left to right (if many of them are candidates for replacing). If S is entirely matched, then the syntax check terminates successfully; otherwise it fails.

LL(1) parsing is a more intuitive approach, since has a strict methodology to convert the grammar rules into running code for the parser. It also runs in $O(n)$ time (n is the length of S). However, when defining grammar rules, left-recursion (that is, rules that use themselves for the definition *in its more left side*) cannot be used, and such grammars need left-factoring (which means that two rules can not begin with common tokens).

However, the G_2 grammar presented above conforms with these two conditions (see section 5.3.2), as it does not have left-recursion in its rules and it does not need left-factoring (as this was previously done for readability and clarity reasons).

Note that *LL(1)* parsing is suitable for the G_2 PDF object grammar defined. Although, as it has been shown in section 5.4.1, the needed lookahead requires three tokens, it is still a *LL(1)* grammar since the extra tokens in the parse window are only used for an isolated situation only. Thus, performance is only slightly affected, occasionally, and the average runtime of parsing is, in practice, as it was a pure *LL(1)* one.

5.4.3 Requirements

This subsection gahters the requirements for the GNU PDF object parser.

Goal

The goal is to develop a PDF object parser tool for the GNU PDF Library. The parser shall not include the final AST structure, because it is being developed in parallel by other GNU PDF contributors; but a prototype for it could be provided. The parser shall not include error recovery routines, because they must be integrated in a later stage of the library development; but hints in specific source code locations, suggesting strategies for recovering from errors, could be provided to facilitate further improvements.

Functional requirements

The following is a list of functional requirements to be achieved by the parser utility:

- The parser utility shall provide PDF object parsing with a number of debugging options.
- The parser utility shall provide PDF object parsing with options for printing and outputting the parsing results.
- The parser utility shall provide a user help interface.
- The parser utility shall provide a quick usage reference.

Non-functional requirements

The following is a list of non-functional requirements to be achieved by the parser:

- The parser shall run in $O(n)$ time (n equals the input stream length).
- The parser utility shall be written in C, conforming with GNU and GNU PDF coding standards [21, 27].
- The parser shall be written from scratch, without directly using any output from a parser-generation tool.
- The parser shall not be written to be directly submitted as a patch for the main development branch of the GNU PDF Library. Moreover, hard work on integration with other parts in development (specially `pdf_object_t` in the object layer) must be performed before submitting parser code to the `pdf-devel` mailing list for reviewing. Getting the parser ready for an official patch is out of the scope of this FDP.
- Source code must be well documented to facilitate improvements from other GNU PDF contributors.

5.4.4 Specification

This section describes the specification stage for the GNU PDF object parser.

Usage

A brief description of the use cases available in the parser utility is presented in this section through the available options of the PDF object parser utility. In this kind of program, a usage guide like this makes more sense than a set of use case diagrams and workflows.

The PDF object recursive descendant parser must be compiled after using it (see section 2.5.2 in chapter 2):

```
$ gcc -lgnupdf rec-pred-parse.c -o rec-pred-parse
```

After doing this, it can be invoked without parameters to get a quick usage tips:

```
$ ./rec-pred-parse
Usage: ./rec-pred-parse [options...] input.pdf
```

Parsing can be achieved by just typing the input PDF file to be parsed. However, it is also possible to check the rest of available options:

```
$ ./rec-pred-parse --help
GNU PDF Library PDF Object Parser v0.01
---------------------------------------
Usage: ./rec-pred-parse [options...] input.pdf

Available options:
  -h,--help                Show this help
```

```
--version                Show version number
-v                       Verbose mode
-w,--win                 Show parsing window
-e                       Show gnupdf lib errors
```

This gives a better perspective of the options available. The more interesting are `-v`, `-w` and `-e`. `-v` enables the verbose mode, and the user can read how functions corresponding to grammar rules are entered and exited. `-w` or `--win` showsh the parsing window tokens each time a token is matched. Finally, `-e` shows messages and errors sent by the rest of used modules of the GNU PDF Library.

5.4.5 Design

This section summarizes the design decisions chosen to achieve the main goal and meet functional and non-functional requirements.

Parsing window

The first issue considered was the *parsing window*. As discussed before (see section 5.4.1), the needed parsing window consists of three tokens at a time: the usual token, and two additional tokens to have a prevision on indirect references. The GNU PDF Library module that provides tokens as the input stream is being read is the *tokeniser module*. This module provides both a data type, `pdf_token_reader_t`, and a function, `pdf_token_read`, that read the input stream from its current position, return the next token (a `pdf_token_t` data type instance) recognized, and wait at a new position for a new token request. With this in mind, designing the parsing window required a single instance of the `pdf_token_reader_t` data type. This token reader reads the three first tokens of the input stream in its initialization, in order to prepare the first valid parsing window. Four `pdf_token_t` instances are stored: three are used to hold the parsing window contents, plus one extra token to allow the parsing window to unget a token (step backwards).

Flags

The parser utility was also designed with some *flags*. These flags are used to control *debug*, *parse window output* and *library messaging*. The user can specify the behavior of the parser with respect these, using appropriate parameters in the parser utility invocation. With the debug flag set on, information about function entrance and return is shown in `stdout` while parsing. With the parse window output flag set on, the parsing window is written in `stdout` while parsing. Finally, if the library messaging flag is on, the user reads also in `stdout` several status messages from the underlying GNU PDF Library modules being used while the parsing runs.

Parsing schema

Once the parsing strategy was chosen (see section 5.4.5), an adaptation of the *LL(1)* algorithm had to be designed for the specific case of the PDF objects grammar (the G_2 grammar). Several theoretical sources were studied [6, 4].

Recursive predictive parsers, were the *LL(1)* algorithm belongs, consist of a set of mutually recursive functions, where each function corresponds to a grammar rule. This means that each rule in G_2 is implemented in the parser with one function. In parser generators each rule automatically generates the code of its own function. Here, this code is going to be written by hand, and later improved as described in section 5.4.2, but the same rules of the generic algorithm apply here.

These rules correspond with all possible EBNF expressions that can be found in the right hand side of the each grammar rule (please see [6, 4] for *nullable*, *first* and *follow* sets definition). These expressions may consist of:

1. A sequence of ORed expressions: $e_1|e_2|...|e_n$.
2. A sequence of concatenated expressions: $e_1e_2...e_n$.
3. A starred expression: e_1*.
4. A plused expression: e_1+.
5. An optional expression: $e_1?$.
6. A non-terminal: A.
7. A terminal: a.

The code to write for each expression is strict (*i.e.* the code to write for a starred expression, $(e_1)*$, is a `while` loop which condition depends on the current lookahead and the *first* set of e_1).

5.4.6 Implementation

This section describes the implementation work on the GNU PDF object parser proposal, considering the information presented in previous sections.

`rec-pred-parse.h`

This is the PDF object parser header. It contains:

1. Included shared libraries and interfaces.
2. Variables and data types for the parser.
3. Headers of functions implementing the parser tool.

The only shared library being included is the GNU PDF Library, accessed by the `pdf.h` interface available in the developing system after compiling and installing GNU PDF Library.

Variables and data types, and its usage description, are splitted into several groups:

- *Stream* variables. These instances are necessary to access the PDF file contents that the user wants to be parsed. There is only one file to parse, but since three token readers are needed to implement the parsing window, three different streams (on the same file) have to be created.

- *Token reader* variable. This instance contains the true tokeniser work, that is, the lexical processor that splits the characters inside the streams following the lexical rules of defined tokens (see table 5.1).
- *Token* variables. Three of them store the current parsing window (*LA(0)*, *LA(1)* and *LA(2)*, whilst the fourth holds the previous token before those three in the parsing window, in order to allow the unget operation.
- *Status* variables. These instances contain statuses, updated as returning values of many functions of the GNU PDF Library. Taking track of them makes debugging more comfortable.
- A *stream length* variable. The parser needs to store the last value of a **Length** key found on a stream dictionary. This is not a sctricly syntactic process (it is, in fact, more likely to be on a semantic processor), but absolutely necessary to correctly process stream contents. As described in section 5.2.7, stream contents can have an arbitrary length. This stream contents, usually encoded (and thus consisting of binary data instead of characters), can not be processed by the tokeniser module. Moreover, they have to be sent to the object layer for later processing. As it will be shown later, since `pdf_obj_t` data type is still under development, the current behavior is to skip stream contents. However, skipping stream contents also requires to know the number of bytes to skip, which is the value hold in this variable.
- Some *flags* variables. These flags control de behavior of running options like the verbose mode, the output of the parsing window or showing library messages (usually error codes and its corresponding descriptions).
- A *version* variable, containing the version number of the parser utility.

Functions are also splitted in functional groups as follows:

- *Grammar rules functions* that implement the parsing *LL(1)* algorithm itself:
  - `pdf_objects`
  - `indirect_object`
  - `contained_object`
  - `atomic_object`
  - `array_object`
  - `array_or_dictionary_value`
  - `dictionary_object`
  - `key_value_pair`
  - `stream_object`
- *Parsing window* functions, that implement desired behavior in the parsing window submodule:
  - `init_parse_window`, which initializes the parsing window and makes it hold the current input stream token (*LA(0)*), its next (*LA(1)*) and the next of its next (*LA(2)*).

  - `MATCH`, which is a convenience method required by the parser to effectively accept a token.
  - `get_token` gets the next token in the input stream. It advances the parse window one position.
  - `unget_token` steps backward one position in the input stream. The parse window returns to the state it had before the last `get_token` operation.
  - `print_parse_window` prints the current tokens in the parse window in `stdout`.
- Several *auxiliary* functions that manage the shell interface menus, and deal with non-PDF objects skipping.

`rec-pred-parse.c`

This file implements the PDF object parser.

After some shell interaction control, the `main` function initializes the *streams* and *token readers*. After this, it calls `init_parse_window` to initialize the parse window, and calls the first rule of the grammar through function `pdf_objects` to begin the file contents processing. This is done repeatedly until the *end-of-file* condition occurs. Finally, resources are deallocated and the program terminates.

Functions are then recursivelly called as the input string is read and the grammar rules process it throught the *grammar rules functions*. These are implemented stricly following the seven rules of the *LL(1)* algorithm (see section 5.4.5). However, as some processing did not match exactly the parsing algorithm, some remarks on modifications performed follow.

Storing stream length During the first attempts, storing the last stream length read was tried to be avoided. Since the stream contents are delimited with the `stream` and `endstream` keywords, it was originally suposed that this could be enough to skip the stream contents without using the explicit stream length. However, these attempts failed, because in fact the keyword `endstream` can appear inside the stream content. This means that this keyword does not really work as a stream content delimiter. After this, read and usage of the stream content length value was implemented as follows in the `key_value_pair` function / rule:

```
if (pdf_token_get_type (token) == PDF_TOKEN_NAME)
    {
      /* When processing a key-value pair, we must check if
       * the current key is the /Length name. In that case,
       * the corresponding value (hold in LA(1) before
       * matching the key) must be saved to correctly process
       * a probable following stream content. This is not a
       * pure syntactic parsing step, but totally necessary.
       */
      if (strcmp (pdf_token_get_name_data (token), "Length") == 0)
        {
          if (pdf_token_get_type (token_la2) == PDF_TOKEN_INTEGER)
            {
              printf("Syntax error: stream lengths specified by
                      indirect references are not supported.\n");
```

```
          }
        else
          {
            stream_length = pdf_token_get_integer_value (token_la1);
          }
      }
```

While reading a dictionary, the parser checks if a key named **Length** is present and, if it is, stores its value. Note that a little trick involving *LA(2)* is used here. This is because sometimes the length is not directly expressed as an integer object; instead of this, it is pointed elsewhere with an indirect reference, which points to some other PDF object in the file truly containing the integer object value for the length. Since this requires processing of the cross-reference table, it is (for the moment) an unsupported feature.

Skipping non-PDF object contents The parser may find some non-PDF objects in the PDF file specified by the user. In the worst case, all file contents may be non-PDF, and then no parsing at all has to be performed. In this case, the function `skip_non_pdf_objects` skips all file contents that do not fit with valid G_2 input:

```
if (debug_window) print_parse_window ();

  while (pdf_token_get_type (token) != PDF_TOKEN_INTEGER
             || pdf_token_get_type (token_la1) != PDF_TOKEN_INTEGER
             || (pdf_token_get_type (token_la2) != PDF_TOKEN_KEYWORD
                 || strcmp (pdf_token_get_keyword_data (token_la2),
                                                        "obj") != 0))
    {
      /* More objects can be contained in the trailer,
       * header, etc. so we skip tokens until a new
       * indirect object is inside the LA window */

      stat = pdf_token_read (reader, 1, &token);
      stat_la1 = pdf_token_read (reader_la1, 1, &token_la1);
      stat_la2 = pdf_token_read (reader_la2, 1, &token_la2);

      if (stat != PDF_OK || stat_la1 != PDF_OK || stat_la2 != PDF_OK)
        {
          if (debug_errors) {
            pdf_perror (stat, NULL);
            pdf_perror (stat_la1, NULL);
            pdf_perror (stat_la2, NULL);
          }

          if (stat == PDF_EEOF || stat_la1 == PDF_EEOF
              || stat_la2 == PDF_EEOF)
            {
              break;
```

```
            }
        }
    }
```

AST and error recovery calls As stated before, two important features of parsing are only partially implemented: AST construction, and error recovery.

AST construction is the final result of the parsing. While the *LL(1)* parser checks the syntactic structure of PDF objects in its input, it must generate a data structure representing the PDF object hierarchy. For instance, an *indirect object*, which is typically the most general, high level PDF object found in PDF files, may contain several nested PDF objects, like dictionary objcets, containing array objects, containing more dictionary objects, and so on. Actually, this hierarchy is the output artifact of the parsing process (see section 5.1).

During the development process of the PDF object parser, this data structure needed for AST construction (the `pdf_obj_t` data type, and associated functions) was being developed in parallel. When the parser development finished, it was still in an early state, and it was decided to prototypically indicate in particular lines of the source code the exact calls that should be performed to successfully generate this AST. As explained in section 5.3.3, the higher and more consistent set of rules of G_2 helped a lot to appropriately place these calls.

Error recovery is a key feature of parsing. Without error recovery, two non-desirable situtations would arise in any parsing operation:

- When facing a parsing error, the parsing routine would return the point where the PDF grammar was not matched, and it would terminate. Hence, no valid nor complete AST would be produced, forcing the user to correct the syntax mistake and re-run the parser again. This is specially undesirable for PDF final users, because usually they do not know almost anything about the PDF format, neither about hacking PDF files. The maximum number of errors should be recovered, even having consequences on PDF rendering, before returning erroneus AST or syntax error messages to the user.

- When facing a parsing error, recoverable or not, the parser shall not stop. Moreover, it should keep track of the bad syntax found, but shall go straight ahead and try to read the rest of the PDF file.

Error recovery shall be implemented in the PDF object parser in a later and more mature stage, when `pdf_obj_t` integration is successful.

5.4.7 Testing

During the development of the parser, some test files containing PDF objects were used. These files followed an incremental approach; for instance, the first PDF file that was tried to be parsed consisted simply of the following contents:

```
1 0 obj
(Hey)
endobj
```

The set of input tests that were used are the following (available online [46]):

1. **test0** tests a PDF indirect object alone.

2. **test1** tests several objects in a single file, covering:
   - Direct objects: boolean, integer, literal strings, hexadecimal strings, name objects, arrays and dictionaries.
   - Indirect objects: tests direct objects nested inside indirect objects.
3. **test2** tests real content from a PDF file. Thus, all objects are indirect objects, which contain: comments, dictionaries, indirect references, indirect references with integers inside arrays (which tests the *LL(3)* rule) and non-encoded stream objects.
4. **stream.pdf**, **stream2.pdf** and **stream3.pdf** test several indirect objects containing encoded stream contents.

On the other side, complete and public PDF files were also successfully parsed. It has to be noted that only PDF objects of those files were parsed, because the rest of the PDF file contents is discarded with the `skip_non_pdf_objects` function. Still non supported features, as well as further work on PDF objects parsing, is detailed in chapter 8.

Chapter 6

Documentation

Many documentation and other community efforts in a number of mediums have been done during the development of this FDP. This chapter describes these efforts.

The chapter is organized as follows. First, the documentation work performed in GNU PDF contributions, mainly in source code and library manuals, is described. Second, articles written to spread the GNU PDF initiative in blogs and social networks are presented. Third, supplied documentation on free and public databases, like the GNU PDF wiki and Wikipedia, is shown. Finally, other work involving helping the GNU PDF community is depicted.

6.1 Source code documentation

The most low-level doumentation written during the FDP has been contributed in the source code files. GNU PDF Hacker's guide [27] has valuable tips on code documentation, such file header structure or function naming conventions.

Additionally, almost each step or non-trivial operation developed has its corresponding explanation note in the source code.

Examples

```
/* At this point all token readers point to the
 * 'endstream' token. Since they have to be moved ahead
 * like in the initial state, we call init_parse_window
 * just like then.
 */
init_parse_window ();

/* Initialize LA(0), LA(1) and LA(2) token readers */
init_parse_window ();
```

6.2 `gnupdf.texi`

`gnupdf.texi` is a key file in the GNU PDF Library source code tree. It contains the GNU PDF Library Reference Manual [28]. It is written in the *texinfo* format. Texinfo [37] is a typesetting syntax used for generating documentation in both on-line and printed form

(creating filetypes as `dvi`, `html`, `pdf`, etc., and its own hypertext format, `info`) with a single source file. It is implemented by a computer program released as free and open source software of the same name, created and made available by the GNU Project from the Free Software Foundation [69].

The GNU PDF Reference Manual has been updated with some contributions, the most important consisting of the API modifications performed as a result of developing the UUID module (see chapter 3).

6.3 Portable Document Features: A GNU PDF developer's blog

When this FDP began, a blog was opened to track the development of contributions: **Portable Document Features, A GNU PDF developer's blog** [48]. The blog features articles with information about the GNU PDF project, developments documented previously in this report, and links to important news in the GNU and PDF worlds.

Since it was opened on October 2010, it has been visited 1,456 times, with an average of 6.45 visitors per day (see figure 6.1). These visits came mainly from Spain, United States, Germany, Switzerland and India (see figure 6.2). Visits came mainly from `planet.gnu.org` referrals, because some news and links were posted there just after the blog opened. Google, `linuxgnublog.org` and Facebook also sent a high number of visitors. The most visited post of all time is *Install GNU PDF library from source* [47], a tutorial guide for newcomers and all interested users to install the GNU PDF Library easy and quick on their systems.

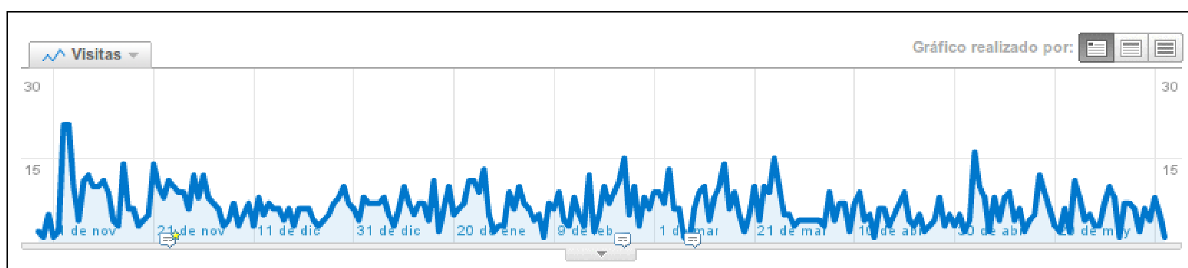


Figure 6.1: Visits to the Portable Document Features blog, from October 2010 to June 2011 (figure provided by Google Analytics).

6.4 Social Networks

Some spreading work has been done also on social networks, specially on Twitter. Relevant content and news for the GNU PDF project was tweeted under the `gnupdf` or `pdf` hashtags. All readers of *Portable Document Features* can also tweet directly references to articles by clicking on the corresponding button in the article page.

6.5 GNU PDF Knowledge Database

Besides the GNU PDF Library and the Juggler, the GNU PDF project also wants to provide free documentation about PDF technology [31]. This documentation is generated in a wiki

	Nivel de detalle: País/territorio	Visitas ↓	Páginas/visita	Promedio de tiempo en el sitio	Porcentaje de visitas nuevas	Porcentaje de rebote
1.	Spain	217	1,87	00:02:26	42,86%	70,97%
2.	United States	216	1,45	00:01:26	82,41%	79,17%
3.	Germany	194	1,29	00:00:37	60,82%	84,54%
4.	Switzerland	121	1,20	00:00:26	23,14%	88,43%
5.	India	69	1,28	00:02:41	59,42%	84,06%
6.	France	68	1,37	00:00:24	89,71%	77,94%
7.	Canada	62	1,47	00:02:17	69,35%	82,26%
8.	Italy	61	1,62	00:02:36	77,05%	59,02%
9.	United Kingdom	49	1,84	00:00:43	87,76%	71,43%
10.	Belgium	40	1,30	00:01:25	22,50%	80,00%

Figure 6.2: Visits to the Portable Document Features blog, from October 2010 to June 2011, with respect to the visit origin (figure provided by Google Analytics).

style, and it is known as the GNU PDF Knowledge Database [23]. From the GNU PDF project page [31]: *As we develop the GNU PDF Library we are generating free documentation about many parts of the PDF specification, the several existing PDF standards, PDF implementation issues, etc. That documentation is available in this webpage and organized into the PDF Knowledge Database.*

Many contributions have been made to this database during the development of this FDP. The most important are located under the article *PDF standards comparison* [5], which is a good summary of all the work performed in chapter 4. This standards comparison initiative has generated a remarkable interest: since April 2011, it has received 3,159 visitors (see figure 6.3). This means that the development of free PDF technologies seems attractive to the current PDF user community, which have been looking for years a free replacement for Adobe SDK (that is, a certified PDF library that complains with official standards and specifications).

Figure 6.3: Visitors to the GNU PDF Knowledge Database article *PDF standards comparison*, which compares the requirements found in some of the current, most used PDF standards and specifications.

6.6 Wikipedia contributions

Some contributions have been also made to Wikipedia [71], most of them in the PDF article [66]. These contributions consisted of making some corrections, adding technology information and improving language in explanations.

6.7 IRC community

The development of this FDP has been performed always standing connected to the `#pdf` channel at `irc.freenode.net`. This was an unvaluable experience regarding two key issues of the Free Software movement: to help others, and to obtain help from others.

On one hand, obtaining help was almost always related to programming the GNU PDF Library. Regarding this, the GNU PDF community has some priceless hackers that, as contributors and volunteers, always have time to answer almost any programming, PDF, or project management question. Without their help, this FDP (and surely neither the GNU PDF project) could not be possible.

On the other hand, as a newcomer there are less oportunities to give help than to obtain it; but, however, there was a few of them. It is common that final users, instead of contributor programmers, enter the channel (or the mailing list) and ask some question regarding PDF. Assistance to these final users has been provided during the development of the project.

Chapter 7

Budget and execution

This chapter describes the budget and the execution of the project, as well as some data and graphics regarding the execution of the project.

7.1 Budget

This section describes the budget of the project.

7.1.1 Software costs

No software licensing costs can be attributed to the project. The entire project is developed using Free Software only.

7.1.2 Hardware costs

Required hardware consists of a desktop computer, a laptop, and a DSL connection during four months (see table 7.1).

Item	**Price (€)**	**Units**	**Cost (€)**
Desktop computer	1,598.00	1	1,598.00
Laptop	525.00	1	525.00
Montly DSL subscription	49,95	4	199.80
		Total	2,322.80

Table 7.1: Budget for hardware costs.

7.1.3 Staff costs

Required staff consists of a programmer (see table 7.2). Price is given in a programmer per hour salary basis, whilst units regard the total number of hours planned for the project. As shown in section 7.2, the total working hours of the project were 624.

7.1.4 Final cost

The total budget of the project is shown in table 7.3.

Item	Price (€)	Units	Cost (€)
Programmer	35	624	21,840.00
		Total	21,840.00

Table 7.2: Budget for staff costs.

Item	Cost (€))
Software	0
Hardware	2,322.80
Staff	21,840.00
Total	24,162.80

Table 7.3: Budget for the project.

7.2 Execution

This section describes how the planning, timing and scheduling was finally run during the project.

The Gantt chart representing the execution of this FDP, including all project stages and substages, is shown in figure 7.1.

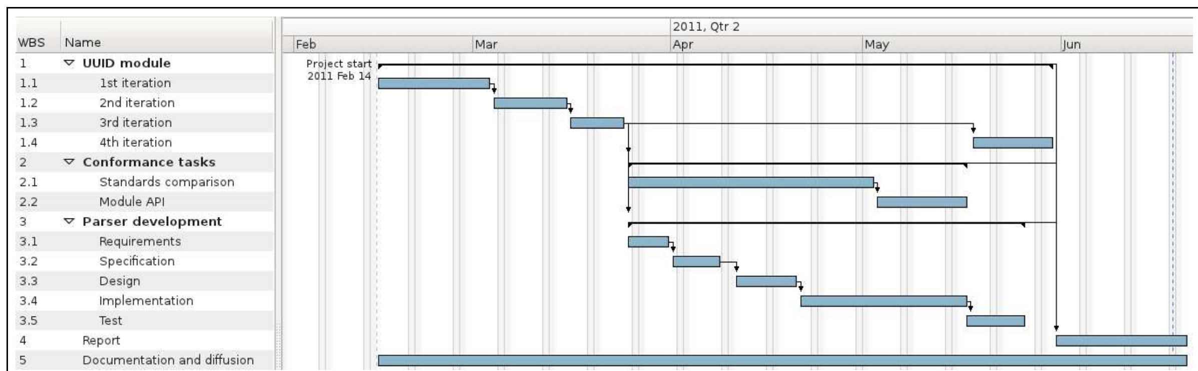


Figure 7.1: Gantt chart of the project.

The project was organised in five work packages: the UUID module, the conformance tasks, the development of the parser, the writing of this report, and documentation and diffusion.

As explained in chapter 2, section 2.6, the UUID module was the first task of the project, and thus it started at the same date of the project start. Once it was developed enough, though not completed, conformance and parser tasks began. For this reason, conformance and parser development had a dependency on the UUID being almost finished. The parser development took some more time resources than the conformance tasks, and thus the final stages of the parser were performed after finishing the conformance issues.

Each work package consisted of a number of subtasks, which are detailed below.

7.2.1 UUID module

The UUID module consists of four iterations, as shown in figure 7.2.

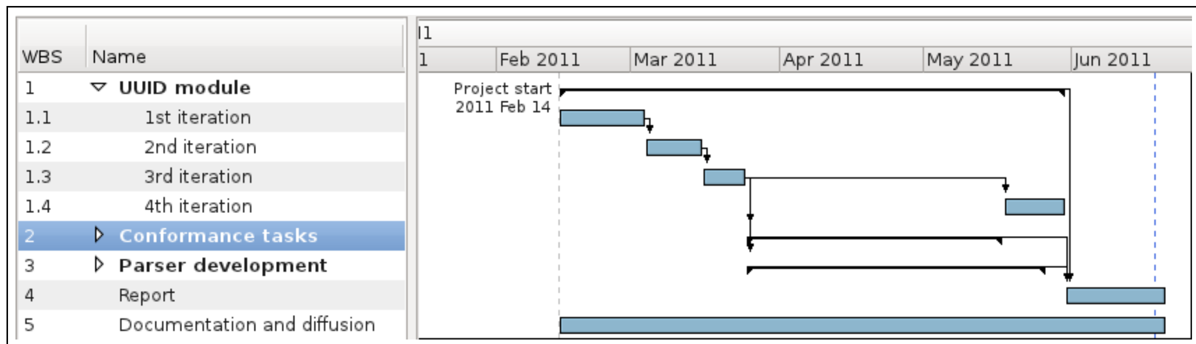


Figure 7.2: Gantt chart of the UUID module.

Each iteration reviews the results of the previous one, improving these results through the patch submitting and reviewing method (see chapter 2, section 2.5.4). However, critical stages like requirements analysis were performed only on the first iteration. Additionally, the first iteration also included specification, design, implementation and testing, and this overall approach made it require more developing time than the rest.

After the third iteration, the UUID module was in a stage advanced enough to allow starting the development of the conformance tasks and the parser. Later, the fourth iteration completed the UUID module.

7.2.2 Conformance tasks

The conformance tasks consisted of two stages: the comparison between an available standard (PDF/A, [11]) and an Adobe specification (PDF 1.4), and the specification of a new module in the base layer of the library, the conformance module. The schedule for these tasks is shown in figure 7.3.

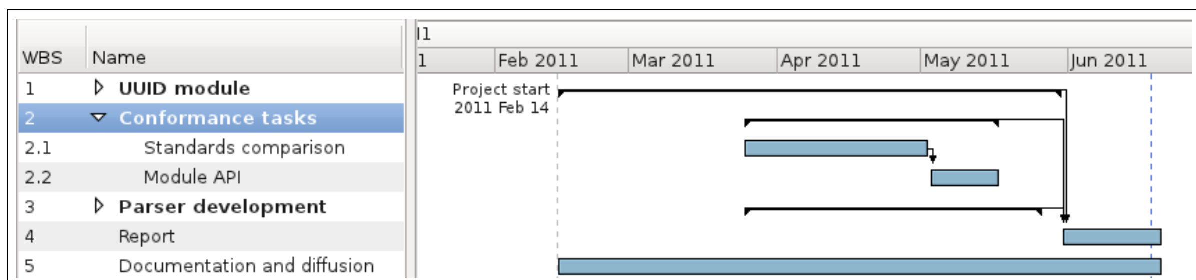


Figure 7.3: Gantt chart of the conformance tasks.

7.2.3 Parser development

The parser development comprised five stages: requirements, specification, design, implementation, and test, as shown in figure 7.4.

1. **Requirements**. In this phase the requirements for the PDF object parser were analysed.

2. **Specification**. In this stage the API of the parser was constructed, as well as the necessary data types, modules and functions were studied. The available options for

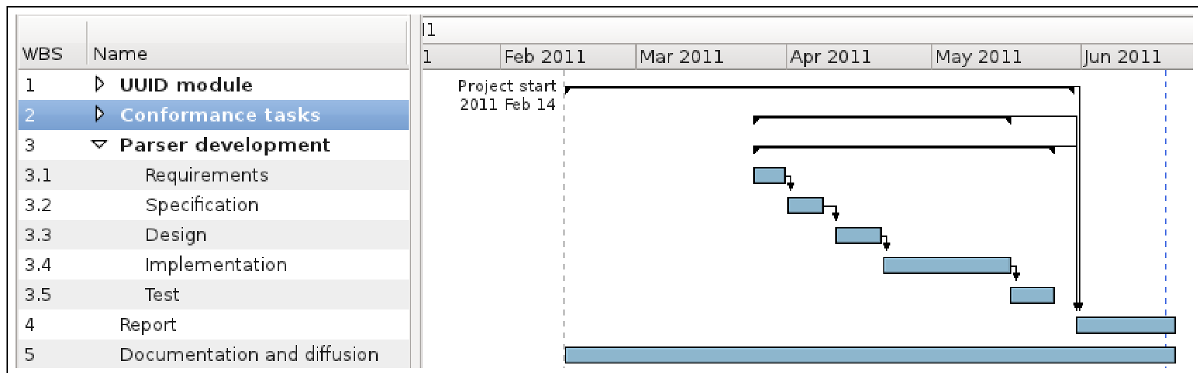


Figure 7.4: Gantt chart of the parser development.

the user of the parser utility were also introduced in this stage.

3. **Design**. This subtask consisted of important decisions regarding the parsing strategy or algorithm, some consequences of the specification phase, such as the availability of several options for the parser tool user, and the design of the parse window.

4. **Implementation**. In this task the parser was coded according with decisions made in the previous stages.

5. **Test**. Finally, the parser needed to be tested to check if valid output was being generated when valid input was provided.

7.2.4 Report and documentation

The fourth and fifth work packages of the project comprised the writing of this report, and the generation of documentation and diffusion articles to spread the GNU PDF initiative. Figure 7.5 shows when these tasks were performed.

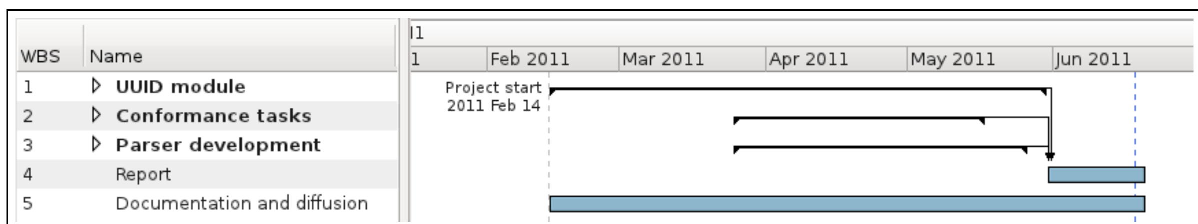


Figure 7.5: Gantt chart of report and documentation writing.

Documentation and diffusion efforts were done during the entire development of the collaboration. Report writing could only be done when all previous work packages (that is, the UUID module, the conformance tasks and the parser developmnet) were completely done.

7.2.5 Overall

All working hours were properly documented during the FDP. As showing the complete records would result in a huge table, a more graphical depiction is presented in figure 7.6. The chart shows that the highest amount of time was dedicated in during April, May and June 2011, because more area is enclosed under the line in these dates. Despite this, the

project efforts began on August and September 2010. The average dediaction per day was 5.47 hours, which is almost one hour higher than planned. The lowest dedication day consisted of 1 working hour, whilst the most intesive day consisted of 13 working hours. The count of working days was 114. Data shown in table 7.4 depicts how these hours were spent.

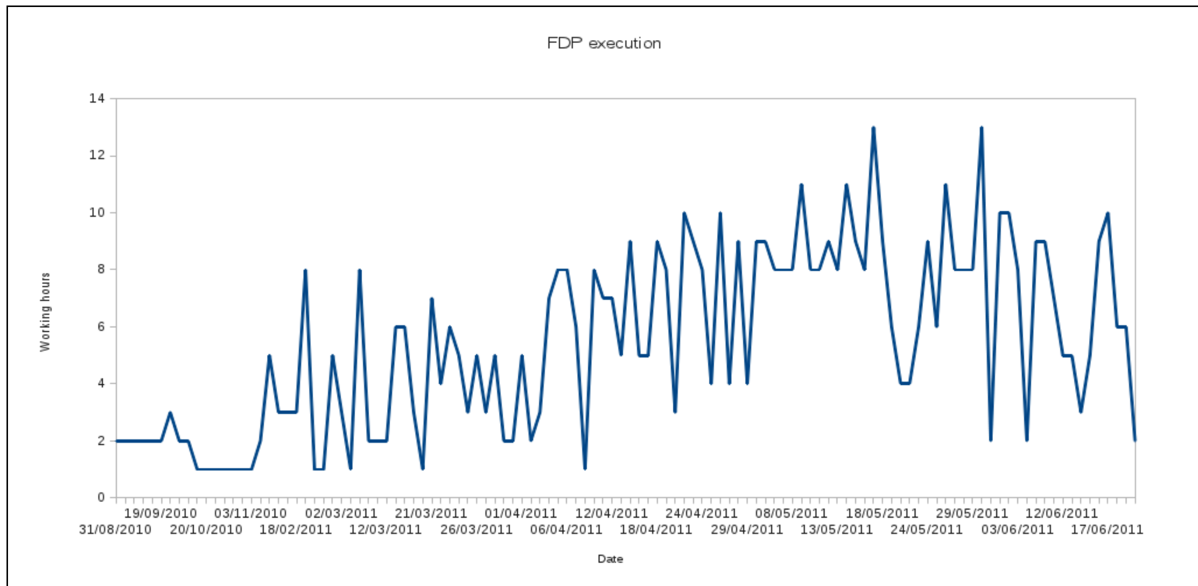


Figure 7.6: Chart of working hours used during the FDP.

Task	**Hours**
UUID module	109
Conformance tasks	99
Parser development	163
Report	148
Documentation	20
Meetings	40
Project management	45
Total	624

Table 7.4: Time spent by task.

Chapter 8

Conclusions

This chapter gathers the conclusions of the project, as well as some further work.

8.1 Goals achieving

Generals goals listed in chapter 1 have been achieved completely. The process of learning how to collaborate, interact, contribute and work with and for the GNU PDF community has been fully learned. While contributing in free software frictions may occur between volunteers, because everyone loves writing free software and is contributing in an altruistic manner, and implicitly wants his or her work to be recognized by peers. The hackers communities behavior has been studied at great length [45]. However, the development of this FDP has been successful in this sense too. All the GNU PDF commuinty is very kind, grateful and willing to help others. No discussion went outside of its matter regarding the project work, programming or anything else, though obviously disagreements about programming happened sometimes. The GNU PDF community is a very comfortable place to work mainly thanks to the efforts and the excitement of José E. Marchesi and Aleksander Morgado.

Contributing the GNU PDF Library with source code was also achieved. As the `ChangeLog` of the library show nowadays, patchs sent by the author were successfully applied to the main branch, providing improvements in the base layer source code, the documentation, and the build system. However, much of the effort reported in this FDP has not been integrated (yet) to the main branch. The reason is that the conformance module and the PDF object parser were developed outside of the common framework of GNU PDF contributions (see chapter 1, section 1.2). This code is being currently reviewed and refactored to adapt it to the GNU PDF tasks management.

The study of the PDF language, standards and specifications has been also successful. At the beginning, dealing with the PDF mess was a difficult task to do. Even the GNU PDF Knowledge Database [23] encouradged volunteers to read, study and understand how and where PDF specifications, standards, subsets and others fit. This has been developed extensively in chapter 4. The PDF language has been understood completely, as well as formatting PDF files and the logical structure of PDF documents. The importance of providing a standard-compliant PDF library has been realised, performing the first steps on standard comparison. A proposal for a compliance module API to achieve the GNU PDF goal of providing standard-compliant facilities has been also given.

The community of GNU PDF developers, as well as general PDF users, has been helped

during the development, mainly in the `pdf-devel` mailing list and `#pdf@irc.freenode.net`. The GNU PDF initiative has been spread through Twitter, the project website [31], and the blog of this FDP [48].

Regarding the specific goals, the UUID module has been provided and applied to the main branch successfully. It took four iterations, as planned in chapter 7. The UUID did not have a significant amount of code, but it does what is needed from it: provides time-based and random-based UUID generation, ASCII representation, and UUID comparison. As explained in chapter 3, implementation did not consist of coding these functionalities from scratch; it consisted of linking an existing library, *libuuid*, which was already providing these functionalities, to the GNU PDF Library, adapting the GNU Build System and writing a layer that maps the GNU PDF Library API with the *libuuid* API. The development of this module helped to better understand the organisation and methodology of GNU PDF, to learn how to develop for a free software project, to apply coding conventions, and to discuss solutions with the rest of the community.

The roadmap for making the GNU PDF Library fully standard compliant has been established. In the second work package, three main goals were achieved:

1. A complete requirements comparison between PDF 1.4 and PDF/A (ISO 19005-1:2005 [11]) was performed (see appendix C).

2. Next, a formalisation of the mapping between conformace levels and requirements was performed and published [5].

3. Finally, as an extension of previous items the API of a conformance module located in the base layer was provided.

A PDF object parser was developed and brought to the community. This parser covers the syntax analysis of any given PDF object with an adapted *LL(1)* algorithm and using an also provided *EBNF LL(1)* grammar (excepting a single rule, which is *LL(3)*), written from scratch directly meeting official ISO 32000-1:2008 [2] specifications. The PDF object parser runs in $O(n)$ time, where n equals the length of the input (the PDF object). The parser development encountered many issues, such as the lookahead requirements that lead to extend the parsing window, and the parallel developing of the required AST structure in the `pdf_obj_t` data type. These and other issues are discused in the further work. Additional difficulties where found while parsing PDF stream objects. In these kind of objects, binary data must be read by the parser, just until the binary content ends with the keyword `endstream`. The issue was that this keyword can appear inside the stream content itself, nullifying its feature of being a delimiter. The solution is to use a property that must be specified in the dictionary of all stream objects: the `Length` key. The value of this key is the total length of the stream, measured in bytes. It is also the valid length for the stream that must be considered by standard conformant readers. With the GNU PDF community support, it was decided to use this value to process properly stream contents. The consequence of doing so is that it enters the domain of the semantic processor: a syntax checker should never consider the values of the tokens read to determine if the input is valid or not; only the arrangement of these tokens should be relevant. However, the weak requirements on streams syntax forced this decision.

This last issue can be compared with the only *LL(3)* rule in the PDF objects grammar, in the sense that both come directly from the PDF language definition. As explained in chapter 5, the lookahead must be extended to three tokens because the syntax of *atomic objects* make

the decision between *integer objects* and *indirect references* ambiguous. This could be solved changing the language itself, maybe introducing an extra keyword or delimiter to express an *indirect reference*, and thus never confusing its first integer with an isolated *integer object*. The same applies for stream contents, where a more strict syntax may not force the reader to use the semantic property `Length`. Surely, these two issues have been identified in the past. However, legacy compatibility with all versions of PDF, including old versions, makes very difficult to modify the base language of PDF objects to correct these points.

Regarding the documentation work, all developments have been appropriately documented during the project, specially in the GNU PDF Knowledge Database, the GNU PDF Reference Manual, and Wikipedia [23, 28, 66].

8.2 Further work

There is still a lot of work to do in GNU PDF regarding semantic and syntax processing.

From a point of view, the proposal on the conformance module can be seen as a particular stage of the overall semantic cheking. Though PDF object syntax is the same for all PDF dialects (and, as a consequence of this, the provided grammar in chapter 5 is valid for any given PDF specification or standard), the *contents* of these objects may be not. As it has been shown in chapter 4, particular standards restrict values of PDF objects in some cases (for instance, a stream dictionary shall not contain the `F` key in PDF/A). These are, in fact, restrictions that the semantic processor must consider. And, as it has been shown [5], restrictions depend on the conformance level that the user wants to apply to the PDF file. Hence, a complete comparison of all requirements found in all PDF speficiations and standards has to be included inside the GNU PDF Library, in order to make it completely standard and specification aware. Regarding this, two major steps shall be done in the future:

1. To improve the provided conformance module API speficiation, with collaboration of all the GNU PDF community. After this, the conformance module should be split in several sub-modules, and the development cycle should begin creating corresponding tasks in the *flyspray* manager.

2. To complete the table provided in the GNU PDF Knowledge Database [23]. This is a key task to do to make the GNU PDF Library fully standards conformant. For the moment, the comparison table compares successfully PDF/A with PDF 1.4. However, the rest of studied specifications and standards shown in chapter 4 have still to be included. This is a hard and arduous task, though very illustrative; everything about the PDF language and dialects can be learnt from these documents. Additionally, this task may be *expensive*: though being official and public, ISO standards have to be previously bought to the organisation, usually at a high price. The Free Software Foundation is specially involved in this, and despite being a slow process, it buys any standard that the developers may need to bring freedom to software.

With a working conformance module, on one hand, and a complete table mapping requirements and conformance levels, on the other hand, the GNU PDF Library will be surely in its way to pass a further standard certification process.

The syntax process can be improved, too. For the moment, PDF objects can be syntactically checked with the developed parser with some restrictions:

1. No AST is returned as output, though the points where and how it should be generated are commented. The AST structure depends on the `pdf_obj_t` data type, which is being developed by another contributor in parallel and it is still not available.

2. The content of PDF stream objects is not being processed. The current behavior is skipping those usually encoded (and octet represented) contents. The same previous reason applies: the content of stream objects must be processed by `pdf_obj_t` functionalities, still under development.

3. Processing the cross-reference table was out of the scope of the PDF object parser. Hence, stream objects whose length is specified with an indirect reference (to another PDF object) are not supported. This feature will be fixed when the PDF file parser (which includes a cross-reference table processor) is developed, because a data type holding the cross-reference table information will let the programmer access data of any referenced PDF object.

4. Error recovering should be performed in a later stage, also when a more complete representation of all the contents of a PDF file is available for the library programmers.

It is clear that these four issues constitute further work that must be developed in the short term. Only two dependencies must be met: the completion of the tasks regarding the `pdf_obj_t` data type, and the development of a PDF file parser (which also needs the `pdf_obj_t` data type to be fully functional).

8.3 Closing remarks

Contributions made in the base layer through the development of the UUID module made the contributor realise that free/open source software projects commonly use their own development methodologies. The process of developing the UUID module arose an easy and effective methodology that is depicted in figure 8.1. This cycle consists of four stages, beginning with coding, which is performed by the contributor alone. When his or her contributions are ready, they are packaged in a *patch*, which is sent to the community (in the GNU PDF case, through the `pdf-devel` mailing list). Then, the community performs the *review*. This can result in two different situations. In the first situation, the patch is refused by some reason. The community discusses the patch contents, and provide tips, advices and improvements to the contributor, in order to polish it. In this case, the subcycle starts again with a new *coding* iteration, where the contributor tries to integrate all community (and his or her own) suggestions. In the second situation, after the *review* the community *accepts* the patch, it is applied to the main development branch, and its improvements are permanently available to the rest of the community.

Despite the quick development of Agile Software Development [56], this iterative, incremental approach differs from methodologies taught by Software Engineering until the last recent years. Differences between classical Software Engineering (SE) methodologies and Free/Open Source Software Development (FOSSD) methodologies have been largely discussed [51], but also recently adopted [50] by SE practices.

It is difficult to evaluate which aspects of the project bore most fruit, but probably the standards comparison (with an increasing visit count, see chapter 6), consisting of the API proposal for a conformance module, and the PDF object parser brought the more interesting

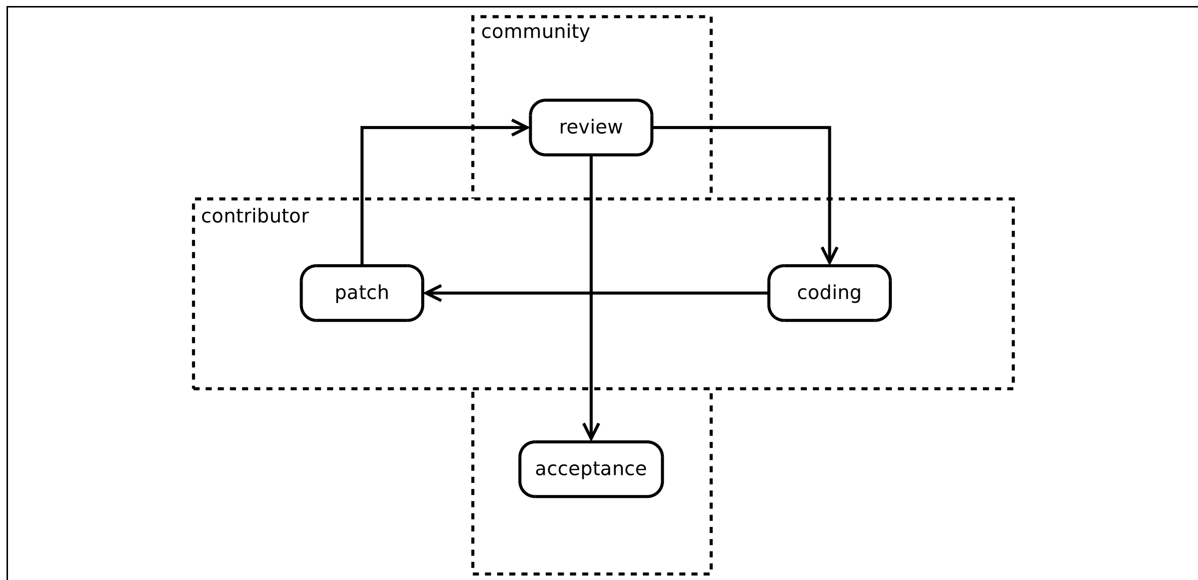


Figure 8.1: The GNU PDF contributing methodology. The horizontal axis contains contributor activities, while the vertical axis contains community activities.

results. The applied methodology worked well, and helped to achieve all goals of the project. However, some planned activities took less time to complete, and some others took more, and thus the schedule had some mismatches between what was planned and how it was executed. If the project had to start again, it should be necessary to assign more hours to the report writing, and some time assigned to the development of the UUID module should be reduced.

Despite this, the project ended successfully from almost all points of view, achieving its goals and, more important, contributing the GNU PDF project with key features such basic types support, conformance awareness and syntax processing. However, there is still a lot of work to do. And this work does not only point to semantic or syntactic processing. The GNU PDF Library was created following the same architecture than the Adobe SDK, in an intent to facilitate users transition from current privative conformant PDF tools, to a true free software support for PDF technologies. This architecture is far from being complete and, though slow, firm and consistent steps are being constantly done to achieve the GNU PDF main goals. In the low term, the base layer (including the conformance support) and the object layer should be fully functional. This will allow programmers to process the whole contents of PDF files, storing them in appropriate data structures in memory, and processing them properly. In the mid term, the most interesting layers from the external library programmer, as well as the external applications using the GNU PDF Library, is to develop widely the document and the page contents layers; that is, finishing the main work with the GNU PDF Library. When this stage arrives, it will be possible to state that PDF technologies are supported by Free Software; but not before. In this ideal situation, that will surely come, the whole community will be able to adapt their programs and applications to a free, high-quality, complete and portable PDF library, certified to be conformant with official PDF standards and, consequently, providing the same functionalities the Adobe SDK does. This scenario sets up the long term for the GNU PDF project, which plans to develop its own PDF suite, the **GNU Juggler**. The GNU Juggler will be a full-fledged PDF viewer and editor making use of the GNU PDF Library, providing final users, industries and governments with all standard

PDF technology and functionalities.

Hence, in this hopefully near future the GNU PDF project would have helped users and the whole society to be more free. For example, users would have the guarantee that the PDF forms used by public administrations are being read and written with transparent, open, accessible, conformance-aware and high-quality software. Enthusiastic readers of eBooks would have their devices operating with free software only, which ensures them to be using efficient, well ported, standard compliant and freedom respectful software to read their favourite books. The efforts of the Free Software community, as well as the excitement and altruism of GNU PDF volunteers and contributors, are putting the GNU PDF project in the right way to bring full freedom and conformance to PDF implementations to the computing world.

Bibliography

[1] Inc. Adobe Systems. Creating PDF/E-ready files. http://www.adobe.com/enterprise/pdfs/pdf-eready-guide-ue.pdf, 2008. [Online; accessed 5-May-2011].

[2] Inc. Adobe Systems. Document management - Portable document format - Part 1: PDF 1.7. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf, 2008. [Online; accessed 20-October-2010].

[3] Inc. Adobe Systems. Extensible Metadata Platform (xmp). http://www.adobe.com/products/xmp/, 2011. [Online; accessed 21-June-2011].

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] Inc. Albert Meroño Peñuela, Free Software Foundation. PDF standards comparison. http://www.gnupdf.org/Standards_comparison, 2011. [Online; accessed 14-May-2011].

[6] José Miguel Rivero Almeida. Syntactic analysis notes. Compilers, FIB-UPC. http://www.lsi.upc.edu/~rivero/cl.html, 2011. [Online; accessed 11-May-2011].

[7] Fredrik Hugosson Arien Malec, Chris Pickett and Robert Lemmen. Check: A unit testing framework for C. http://check.sourceforge.net/, 2011. [Online; accessed 16-June-2011].

[8] Valgrind Developers. Valgrind. http://valgrind.org/, 2011. [Online; accessed 20-March-2011].

[9] Facultat d'Informàtica de Barcelona. Degree subject curriculum. http://www.fib.upc.edu/en/estudiar-enginyeria-informatica/grau.html, 2011. [Online; accessed 15-June-2011].

[10] Kurt Wall et al. *Linux Programming Unleashed*. Sams, Indianapolis, IN, USA, 2nd edition, 2000.

[11] International Organization for Standardization. Document management. Electronic document file format for long-term preservation. Part 1: Use of PDF 1.4 (PDF/A-1), 2005.

[12] International Organization for Standardization. Document management. Engineering document format using PDF. Part 1: Use of PDF 1.6 (PDF/E-1), 2008.

[13] International Organization for Standardization. Graphic technology. Variable data exchange. Part 2: Using PDF/X-4 and PDF/X-5 (PDF/VT-1 and PDF/VT-2), 2010.

[14] International Organization for Standardization. Document management applications. Electronic document file format enhancement for accessibility. Part 1: Use of ISO 32000-1 (PDF/UA-1), 2011.

[15] Internet Engineering Task Force. RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace. `http://www.ietf.org/rfc/rfc4122.txt`, 2005. [Online; accessed 21-October-2010].

[16] Open Software Foundation. UUIDs and GUIDs. `http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt`, 1998. [Online; accessed 21-October-2010].

[17] Inc. Free Software Foundation. Bison - GNU parser generator. `http://www.gnu.org/software/bison/`, 2011. [Online; accessed 14-Apr-2011].

[18] Inc. Free Software Foundation. Copyright papers. `http://www.gnu.org/prep/maintain/html_node/Copyright-Papers.html`, 2011. [Online; accessed 16-June-2011].

[19] Inc. Free Software Foundation. Free Software Foundation. `http://www.fsf.org/`, 2011. [Online; accessed 15-June-2011].

[20] Inc. Free Software Foundation. The GNU Build System. `http://www.gnu.org/s/hello/manual/autoconf/The-GNU-Build-System.html`, 2011. [Online; accessed 14-May-2011].

[21] Inc. Free Software Foundation. GNU Coding Standards. `http://www.gnu.org/prep/standards/`, 2011. [Online; accessed 15-May-2011].

[22] Inc. Free Software Foundation. GNU PDF Goals and Motivations. `http://www.gnupdf.org/Goals_and_Motivations`, 2011. [Online; accessed 19-June-2011].

[23] Inc. Free Software Foundation. GNU PDF knowledge database. `http://www.gnupdf.org/Category:PDF`, 2011.

[24] Inc. Free Software Foundation. GNU PDF Library API Consistency Report. `http://www.gnupdf.org/prmgt/apic.html`, 2011. [Online; accessed 19-June-2011].

[25] Inc. Free Software Foundation. GNU PDF Library Architecture Guide. `http://gnupdf.org/manuals/gnupdf-arch-manual/gnupdf-arch.html`, 2011. [Online; accessed 16-June-2011].

[26] Inc. Free Software Foundation. GNU PDF Library Cyclomatic Complexity Report. `http://www.gnupdf.org/prmgt/cyclo.html`, 2011. [Online; accessed 19-June-2011].

[27] Inc. Free Software Foundation. GNU PDF Library Hackers Guide. `http://gnupdf.org/manuals/gnupdf-hg-manual/gnupdf-hg.html`, 2011. [Online; accessed 14-May-2011].

[28] Inc. Free Software Foundation. GNU PDF Library Reference Manual. `http://gnupdf.org/manuals/gnupdf-manual/gnupdf.html#UUIDs`, 2011. [Online; accessed 14-May-2011].

[29] Inc. Free Software Foundation. GNU PDF Library Unit Testing Report. `http://www.gnupdf.org/prmgt/ut.html`, 2011. [Online; accessed 19-June-2011].

[30] Inc. Free Software Foundation. GNU PDF pdf-devel mailing list archives. http://lists.gnu.org/archive/html/pdf-devel/, 2011. [Online; accessed 21-June-2011].

[31] Inc. Free Software Foundation. GNU PDF project. http://www.gnupdf.org/Main_Page, 2011. [Online; accessed 18-April-2011].

[32] Inc. Free Software Foundation. GNU PDF Tasks Management. http://gnupdf.org/manuals/gnupdf-hg-manual/html_node/Tasks-management.html#Tasks-management, 2011. [Online; accessed 17-June-2011].

[33] Inc. Free Software Foundation. The GNU Portability Library Reference Manual. http://www.gnu.org/s/hello/manual/autoconf/The-GNU-Build-System.html, 2011. [Online; accessed 27-May-2011].

[34] Inc. Free Software Foundation. High Priority Free Software Projects. http://www.fsf.org/campaigns/priority-projects/, 2011. [Online; accessed 20-September-2010].

[35] Inc. Free Software Foundation. Information for Newcomers. http://gnupdf.org/manuals/gnupdf-hg-manual/html_node/Information-for-Newcomers.html#Information-for-Newcomers, 2011. [Online; accessed 15-June-2011].

[36] Inc. Free Software Foundation. Introduction to PDF. http://www.gnupdf.org/Introduction_to_PDF, 2011. [Online; accessed 14-May-2011].

[37] Inc. Free Software Foundation. Texinfo - The GNU Documentation System. http://www.gnu.org/software/texinfo/, 2011. [Online; accessed 9-June-2011].

[38] Inc. Free Software Foundation. Why assign. http://www.gnu.org/licenses/why-assign.html, 2011. [Online; accessed 16-June-2011].

[39] Freedesktop.org. Poppler. http://poppler.freedesktop.org/, 2011. [Online; accessed 9-June-2011].

[40] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. http://dinosaur.compilertools.net/yacc/, 2011. [Online; accessed 14-Apr-2011].

[41] Canonical Ltd. Bazaar. http://bazaar.canonical.com/en/, 2011. [Online; accessed 17-June-2011].

[42] Unix Software Technologies Open Source Software Project. OSSP uuid. http://www.ossp.org/pkg/lib/uuid/, 2011. [Online; accessed 20-March-2011].

[43] Terence Parr. The ANTLR Parser Generator. http://www.antlr.org/about.html, 2011. [Online; accessed 14-Apr-2011].

[44] Terence Parr. PCCTS 1.33. http://www.antlr2.org/pccts133.html, 2011. [Online; accessed 14-Apr-2011].

[45] Linus Torvalds (Contributor) Pekka Himanen, Manuel Castells (Epilogue). *The Hacker Ethic and the Spirit of the Information Age*. Random House, 2001.

[46] Albert Meroño Peñuela. Home page. http://www.albertmeronyo.com/, 2011. [Online; accessed 21-June-2011].

[47] Albert Meroño Peñuela. Install GNU PDF library from source. `http://blog.albertmeronyo.com/?p=19`, 2011.

[48] Albert Meroño Peñuela. Portable Document Features, a GNU PDF developer's blog. `http://blog.albertmeronyo.com/`, 2011.

[49] Eric S. Raymond. The Cathedral and the Bazaar. `http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`, 2011. [Online; accessed 17-June-2011].

[50] Suzanne Robertson and James Robertson. *Mastering the Requirements Process (2nd Edition)*. Addison-Wesley Professional, 2006.

[51] Walt Scacchi. When Is Free/Open Source Software Development Faster, Beter, and Cheaper than Software Engineering? `http://www.ics.uci.edu/~wscacchi/Papers/New/Scacchi-BookChapter.pdf`, 2004. [Online; accessed 19-June-2011].

[52] Steve Summit. C Programming. `http://c-faq.com/~scs/cclass/int/sx5.html`, 1996-1999. [Online; accessed 12-May-2011].

[53] Theodore Ts'o. e2fsprogs. `http://e2fsprogs.sourceforge.net/`, 2011. [Online; accessed 20-March-2011].

[54] Jr. Allen B. Tucker and Robert E. Noonan. *Programming Languages: Principles and Paradigms*. McGraw-Hill Higher Education, 1st edition, 2001.

[55] International Telecommunication Union. ITU X.667 - Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 object identifier components. `http://www.itu.int/ITU-T/studygroups/com17/oid/X.667-E.pdf`, 2004. [Online; accessed 21-October-2010].

[56] Wikipedia. Agile software development — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Agile_software_development&oldid=433418611`, 2011. [Online; accessed 20-June-2011].

[57] Wikipedia. Buffer overflow. `http://en.wikipedia.org/wiki/Buffer_overflow`, 2011. [Online; accessed 30-May-2011].

[58] Wikipedia. Cyclomatic complexity — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=432737831`, 2011. [Online; accessed 17-June-2011].

[59] Wikipedia. Extensible Metadata Platform — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Extensible_Metadata_Platform&oldid=432523749`, 2011. [Online; accessed 21-June-2011].

[60] Wikipedia. File format — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=File_format&oldid=431357757`, 2011. [Online; accessed 5-June-2011].

[61] Wikipedia. GNU build system — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=GNU_build_system&oldid=422097679`, 2011. [Online; accessed 22-May-2011].

[62] Wikipedia. Lookahead — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Lookahead&oldid=423190618, 2011. [Online; accessed 9-June-2011].

[63] Wikipedia. PDF/E — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=PDF/E&oldid=414801439, 2011. [Online; accessed 5-May-2011].

[64] Wikipedia. PDF/VT — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=PDF/VT&oldid=422625044, 2011. [Online; accessed 1-June-2011].

[65] Wikipedia. PDF/X — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=PDF/X&oldid=427208474, 2011. [Online; accessed 5-May-2011].

[66] Wikipedia. Portable document format — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Portable_Document_Format&oldid=427176991, 2011. [Online; accessed 5-May-2011].

[67] Wikipedia. Reentrant (computing) — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Reentrant_(computing)&oldid=427458204, 2011. [Online; accessed 30-May-2011].

[68] Wikipedia. Requirement — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Requirement&oldid=430638189, 2011. [Online; accessed 3-June-2011].

[69] Wikipedia. Texinfo — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Texinfo&oldid=424910847, 2011. [Online; accessed 11-June-2011].

[70] Wikipedia. Universally unique identifier — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Universally_unique_identifier&oldid=422065783, 2011. [Online; accessed 3-April-2011].

[71] Wikipedia. Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Main_Page, 2011.

Appendix A

FSF licensing

The FSF requires some papers to be signed by contributors of GNU projects. This appendix gathers pictures and reproductions of these documents.

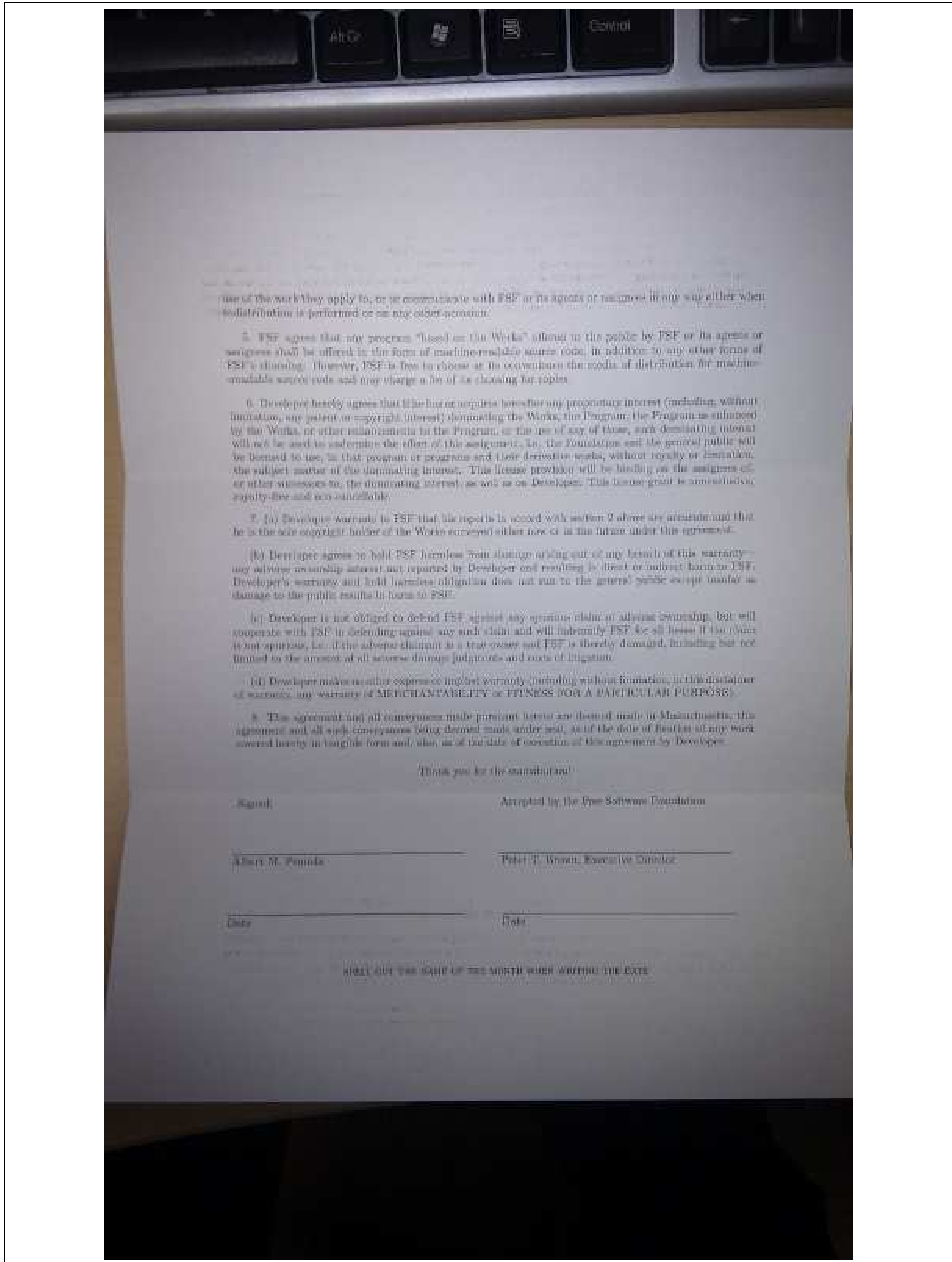
Figure A.1: Envelope of the FSF copyright assignment letter.

Figure A.2: Contents of the FSF copyright assignment letter (sticker included!).

FREE SOFTWARE
FOUNDATION

51 Franklin St, 5th Floor Boston MA 02110-1301 U.S.A
tel: 1.617.542.5942 | fax: 1.617.542.2652 | www.fsf.org

Figure A.3: FSF copyright assignment (1/2).



use of the work they apply to, or to communicate with FSF or its agents or assignees in any way either when redistribution is performed or on any other occasion.

5. FSF agrees that any program "based on the Works" offered to the public by FSF or its agents or assignees shall be offered in the form of machine-readable source code, in addition to any other forms of FSF's choosing. However, FSF is free to choose at its convenience the media of distribution for machine-readable source code and may charge a fee of its choosing for copies.

6. Developer hereby agrees that if he has or acquires hereafter any proprietary interest (including, without limitation, any patent or copyright interest) dominating the Works, the Program, the Program as enhanced by the Works, or other enhancements to the Program, or the use of any of these, such dominating interest will not be used to undermine the effect of this assignment, i.e. the Foundation and the general public will be licensed to use, in that program or programs and their derivative works, without royalty or limitation, the subject matter of the dominating interest. This license provision will be binding on the assignees of, or other successors to, the dominating interest, as well as on Developer. This license grant is nonexclusive, royalty-free and non-cancellable.

7. (a) Developer warrants to FSF that his reports in accord with section 2 above are accurate and that he is the sole copyright holder of the Works conveyed either now or in the future under this agreement.

(b) Developer agrees to hold FSF harmless from damage arising out of any breach of this warranty—any adverse ownership interest not reported by Developer and resulting in direct or indirect harm to FSF. Developer's warranty and hold harmless obligation does not run to the general public except insofar as damage to the public results in harm to FSF.

(c) Developer is not obliged to defend FSF against any spurious claim of adverse ownership, but will cooperate with FSF in defending against any such claim and will indemnify FSF for all losses if the claim is not spurious, i.e. if the adverse claimant is a true owner and FSF is thereby damaged, including but not limited to the amount of all adverse damage judgments and costs of litigation.

(d) Developer makes no other express or implied warranty (including without limitation, in this disclaimer of warranty, any warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE).

8. This agreement and all conveyances made pursuant hereto are deemed made in Massachusetts, this agreement and all such conveyances being deemed made under seal, as of the date of fixation of any work covered hereby in tangible form and, also, as of the date of execution of this agreement by Developer.

Thank you for the contribution!

Signed: Accepted by the Free Software Foundation

Albert M. Pounds Peter T. Brown, Executive Director

Date Date

SPELL OUT THE NAME OF THE MONTH WHEN WRITING THE DATE

Figure A.4: FSF copyright assignment (2/2).

```
DISCLAIMER OF RIGHTS BY A COLLEGE OR UNIVERSITY

  We agree that software and other authored works of the 'Released
Category' (defined below), made by _________________________, a
student or graduate student at this school, prior to the date of this
document, and for _____ years thereafter, are freely assignable by
said student to Free Software Foundation (FSF) for distribution and
sharing under its free software policies.  We disclaim any status as
the author or owner of such works; we do not consider them as works
made for hire for us.

 The Released Category comprises

(a) changes and enhancements to software already (as of the time such
change or enhancement is made) freely circulating under stated terms
permitting public redistribution, whether in the public domain, or
under the FSF's GNU General Public License, or under the FSF's GNU
Lesser General Public License (a.k.a. the GNU Library General Public
License), or under other such terms; and

(b) operating system components for operating systems providing
substantially the same functionality as portions of UNIX, BSD,
Microsoft Windows, or other popular operating systems.
The Released Category excludes ______________________ [if 'none',
please so state; thank you--FSF].
```

Figure A.5: FSF university disclaimer (1/2).

```
  We affirm that we will do nothing in the future to undermine this
release.  If we have or acquire hereafter any proprietary interest
(including, without limitation, any patent or copyright interest)
dominating the works, or the programs to which these works constitute
changes and enhancements, or use of those programs, then the Free Software
Foundation and the general public will be permanently and irrevocably
licensed to use, in these works and in the programs they enhance,
without royalty or limitation, the subject matter of the dominating the
works, or the programs to which these works constitute changes and
enhancements, or use of those programs, then the Free Software
Foundation and the general public will be permanently and irrevocably
licensed to use, in these works and in the programs they enhance,
without royalty or limitation, the subject matter of the dominating
interest.

  We make no warranty as to the quality of the material or as to
the presence or absence of rights therein of any other party, and we
do not purport to disclaim, release or grant any rights other than our
own.

  Given as a sealed instrument this ___ day of ______, 20__.


 signed:___________________________,

       ___________________________, (PLEASE PRINT YOUR NAME)

       ___________________________ (title) of

       ___________________________ (name of institution),
```

Figure A.6: FSF university disclaimer (2/2).

Appendix B

Licensing notes of UUID libraries

B.1 libbuid in e2fsprogs

```
/*
 * gen_uuid.c --- generate a DCE-compatible uuid
 *
 * Copyright (C) 1996, 1997, 1998, 1999 Theodore Ts'o.
 *
 * %Begin-Header%
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, and the entire permission notice in its entirety,
 *    including the disclaimer of warranties.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The name of the author may not be used to endorse or promote
 *    products derived from this software without specific prior
 *    written permission.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ALL OF
 * WHICH ARE HEREBY DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
 * OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
 * USE OF THIS SOFTWARE, EVEN IF NOT ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
```

```
 * %End-Header%
 */
```

B.2 OSSP uuid

```
/*
**  OSSP uuid - Universally Unique Identifier
**  Copyright (c) 2004-2008 Ralf S. Engelschall <rse@engelschall.com>
**  Copyright (c) 2004-2008 The OSSP Project <http://www.ossp.org/>
**
**  This file is part of OSSP uuid, a library for the generation
**  of UUIDs which can found at http://www.ossp.org/pkg/lib/uuid/
**
**  Permission to use, copy, modify, and distribute this software for
**  any purpose with or without fee is hereby granted, provided that
**  the above copyright notice and this permission notice appear in all
**  copies.
**
**  THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
**  WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
**  MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
**  IN NO EVENT SHALL THE AUTHORS AND COPYRIGHT HOLDERS AND THEIR
**  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
**  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
**  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
**  USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
**  ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
**  OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
**  OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
**  SUCH DAMAGE.
**
**  uuid.c: library API implementation
*/
```

Appendix C

PDF/A vs PDF 1.4 requirements

This appendix describes the differences between ISO 19005-1:2005 (from now, refered as *PDF/A* as well), which describes the PDF/A-1a and PDF/A-1b conformance levels, and the non-standardized document *PDF Reference, Third edition* (from now, refered as *PDF 1.4* as well), from Adobe, which describes the PDF 1.4 format, in terms of requirements alignment.

Constraints described in PDF/A restrict the overall capabilities offered by PDF 1.4, in order to ensure that PDF/A compliant files contain features that are stricly needed for archiving purposes only.

The following subsections are equivalent to those described in the ISO 19005-1:2005 document. For each subsection, requirements are uniquely identified by a *requirement identifier*, which follows the format
`[document-reference section paragraph]`
where:

- `document-reference` is a reference for the document that contains the requirement description (*e.g.* ISO 19005-1:2005).
- `section` is a reference for the section in `document-reference` that contains the requirement description (*e.g.* 6.1.3).
- `paragraph` is a list of comma-separated numbers that identify the paragraphs in `section` that contain the requirement description (*e.g.* 1).

For instance, the following requirement for file trailer dictionary

> The file trailer dictionary shall contain the **ID** keyword.

is uniquely identified with *requirement identifier* `[ISO19005-1:2005 6.1.3 1]`.

Note that only non-aligned requirements are shown here. This should not happen because PDF subset standards shall constraint whole PDF specifications (*e.g.* PDF/A vs PDF 1.4). This is not true for all requirements contained in PDF subset standards. Often, PDF subset standards contain requirements that are totally aligned with the PDF specification; sometimes, the difference is just a matter of semantics, as both requirements mean exactly the same (*e.g* concepts such as *end-of-line character* and *new line* are confusingly merged). Thus, PDF subset standards can not, at least formally, be considered as a set of pure restrictions, since they contain both restrictions and redundant, repeated or obvious requirements. These last are not discused in this report.

C.1 File structure

- `[ISO19005-1:2005 6.1.2 2]` The file header line shall be immediately followed by a comment consisting of a % character followed by at least four characters, each of whose encoded byte values shall have a decimal value greater than 127.

  Non-conformance. `[AdobePDF1.4 3.4.1 8]` says that is recommended, but not required: *it is recommended that the header line be immediately followed by a comment line containing at least four binary characters—that is, characters whose codes are 128 or greater.*

- `[ISO19005-1:2005 6.1.3 1,2]` The file trailer dictionary shall contain the **ID** keyword. If linearized, the ID keyword shall be present in both the first page trailer and the last trailer dictionaries and the value of both instances of the keyword shall be identical.

  Non-conformance. `[AdobePDF1.4 3.4.4 4]` says this keyword is optional (and only available from PDF 1.1), but not required.

- `[ISO19005-1:2005 6.1.3 1,3]` The keyword **Encrypt** shall not be used in the trailer dictionary (encryption and password-protected access permissions disallowed).

  Non-conformance. `[AdobePDF1.4 3.4.4 4]` says this is required only if document is encrypted (and only available from PDF 1.1).

- `[ISO19005-1:2005 6.1.7 4]` A stream object dictionary shall not contain the **F**, **FFilter**, or **FDecodeParams** keys.
  Non-conformance. `[AdobePDF1.4 3.2.7 12]` says these keys are optional (and only availabe from PDF 1.2), hence a stream object dictionary may contain them.

- `[ISO19005-1:2005 6.1.10 1]` The **LZWDecode** filter shall not be permitted.

  Non-conformance. `[AdobePDF1.4 3.3.3 1]` allows the use of this filter.

- `[ISO19005-1:2005 6.1.11 1]` A file specification dictionary shall not contain the **EF** key.

  Non-conformance. `[AdobePDF1.4 3.10.2 1]` allows the use of this key (from PDF 1.3); it is required only if key **RF** is present.

- `[ISO19005-1:2005 6.1.11 1]` A file's name dictionary shall not contain the **EmbeddedFiles** key.

  Non-conformance. `[AdobePDF1.4 3.6.3 2]` says this key is optional (and only available from PDF 1.4).

- `[ISO19005-1:2005 6.1.11 1]` The document catalog dictionary shall not contain a key with the name **OCProperties.**

  Non-conformance. `[AdobePDF1.5 3.6.1 2]` says this key is optional (from PDF 1.5),

and required if a document contains optional content. Thus, this requirement is aligned with PDF 1.4, but not with PDF 1.5.

C.2 Graphics

- `[ISO19005-1:2005 6.2.2 1,2]` If a file references Output Intent dictionaries in its **OutputIntents** array (catalog), they must have *GTS_PDFA1* as the value for the **S** key, and a valid ICC profile stream as the value for the **DestOutputProfile** key. If a file's **OutputIntents** array contains more than one entry, then all entries shall have as the value of the **DestOutputProfile** key the same indirect object.

  Non-conformance. `[AdobePDF1.4 9.10.4 4]` says only *PDF_PDFX* is recognised as a valid value for the **S** key; **DestOutputProfile** is not always required.

- `[ISO19005-1:2005 6.2.3.2 2]` A conforming reader shall not use the **Alternate** colour space specified in an ICC profile stream dictionary.

  Non-conformnance. `[AdobePDF1.4 4.5.4 47]` says that using the **Alternate** key is optional.

- `[ISO19005-1:2005 6.2.3.3 1]` A conforming file may use either the **DeviceRGB** or **DeviceCMYK** colour space but shall not use both.

  Non-conformance. `[AdobePDF1.4]` does not restrict this.

- `[ISO19005-1:2005 6.2.3.3 1]` If an uncalibrated colour space is used in a file then that file shall contain a PDF/A-1 OutputIntent.

  Non-conformance. `[AdobePDF1.4]` does not restrict this.

- `[ISO19005-1:2005 6.2.3.3 1]` **DeviceRGB** may be used only if the file has a PDF/A-1 OutputIntent that uses an **RGB** colour space. **DeviceCMYK** may be used only if the file has a PDF/A-1 OutputIntent that uses a **CMYK** colour space.

  Non-conformance. Though `[AdobePDF1.4]` specifies conversion methods, it does not specify conditions of applyability are not.

- `[ISO19005-1:2005 6.2.3.3 2,3]` When rendering a **DeviceGray** colour specification in a file whose **OutputIntent** is an **RGB** profile, a conforming reader shall convert the **DeviceGray** colour specification to **RGB** by the method described in PDF Reference (PDF 1.4). When rendering a **DeviceGray** colour specification in a file whose **OutputIntent** is a **CMYK** profile, a conforming reader shall convert the **DeviceGray** colour specification to **DeviceCMYK** by the method described in PDF Reference (PDF 1.4).

  Non-conformance. `[AdobePDF1.4]` does not restrict this.

- `[ISO19005-1:2005 6.2.3.3 4]` When rendering colours specified in a device-dependent colour space a conforming reader shall use the file's PDF/A-1 OutputIntent dictionary

as the source colour space.

Non-conformance. `[AdobePDF1.4]` does not restrict this.

- `[ISO19005-1:2005 6.2.3.4 1,2]` When a conforming reader renders colour spaces based on **DeviceN** or **Separation** colour spaces, and if the named colourants in the colour space are all from the list **Cyan**, **Magenta**, **Yellow**, **Black**, the file has an **OutputIntent**, and that **OutputIntent** is a **CMYK** profile, then the colourants shall be treated as components of the colour space specified by the PDF/A-1 OutputIntent dictionary, and the alternate colour space shall not be used.

  Non-conformance. `[AdobePDF1.4]` does not restrict this.

- `[ISO19005-1:2005 6.2.3.4 1,3]` When a conforming reader renders colour spaces based on **DeviceN** or **Separation** colour spaces, if the output device does not support the **Separation** colour space or **DeviceN** colourants, then the **Alternate** colour space shall be used.

  Non-conformance. `[AdobePDF1.4]` does not restrict this.

- `[ISO19005-1:2005 6.2.3.4 4]` The **Alternate** colour space of a **Separation** or **DeviceN** colour space shall obey all restrictions on colour spaces (restrictions indicated in `[ISO19005-1:2005 6.2.3.2 2]`, and `[ISO19005-1:2005 6.2.3.3 1]`, `[ISO19005-1:2005 6.2.3.3 1]`, `[ISO19005-1:2005 6.2.3.3 2,3]` and `[ISO19005-1:2005 6.2.3.3 4]`).

  Non-conformance. `[AdobePDF1.4]` does not restrict this.

- `[ISO19005-1:2005 6.2.4 1]` An Image dictionary shall not contain the **Alternates** key or the **OPI** key.

  Non-conformance. `[AdobePDF1.4 4.8.4 3]` allows an Image dictionary to contain these keys, since they are optional (since PDF 1.3 and PDF 1.2 respectively).

- `[ISO19005-1:2005 6.2.4 2]` If an Image dictionary contains the **Interpolate** key, its value shall be false.

  Non-conformance. `[AdobePDF1.4 4.8.4 3]` allows an Image dictionary to contain these keys, since they are optional (since PDF 1.3 and PDF 1.2 respectively).

- `[ISO19005-1:2005 6.2.4 3]` If the Intent key is used in an Image dictionary, then its values shall be one of *RelativeColorimetric*, *AbsoluteColorimetric*, *Perceptual* or *Saturation*.

  Non-conformance. `[AdobePDF1.4 4.5.4]` allows extensions to this set: *Note, however, that the exact set of rendering intents supported may vary from one output device to another; a particular device may not support all possible intents, or may support additional ones beyond those listed in the table.*

- `[ISO19005-1:2005 6.2.5 1,2,3,4]` A form XObject shall not contain the **OPI** key, the **Subtype2** key with a value of *PS*, or the **PS** key.

  Non-conformance. `[AdobePDF1.4 4.9.1 2]` says that the **OPI** key is optional, although keys **Subtype2** and **PS** are not allowed.

- `[ISO19005-1:2005 6.2.6 1]` A conforming file shall not contain any reference XObjects.

  Non-conformance. `[AdobePDF1.4 4.9.3]` allows reference XObjects.

- `[ISO19005-1:2005 6.2.7 1]` A conforming file shall not contain any PostScript XObjects.

  Non-conformance. `[AdobePDF1.4 4.10]` allows PostScript XObjects.

- `[ISO19005-1:2005 6.2.8 1,2]` An ExtGState dictionary shall not contain the **TR** key. An ExtGState dictionary shall not contain the **TR2** key with a value other than *Default*. A conforming reader may ignore any instance of the **HT** key in an ExtGState dictionary. The values of the **RI** key, if used in an ExtGState dictionary, shall be one of *RelativeColorimetric*, *AbsoluteColorimetric*, *Perceptual* or *Saturation*.

  Non-conformance. `[AdobePDF1.4 4.3.4 4]` specifies that these keys are optional in all cases. The **TR2** key may have other values than *Default*. The **HT** value may not be ignored by a conforming reader.

- `[ISO19005-1:2005 6.2.9 1]` Where a rendering intent is specified, its value shall be one of the four values *RelativeColorimetric*, *AbsoluteColorimetric*, *Perceptual* or *Saturation*.

  Non-conformance. `[AdobePDF1.4 4.5.4]` allows extensions to this set: *Note, however, that the exact set of rendering intents supported may vary from one output device to another; a particular device may not support all possible intents, or may support additional ones beyond those listed in the table.*

- `[ISO19005-1:2005 6.2.10 1]` A content stream shall not contain any operators not defined in `[AdobePDF1.4]` even if such operators are bracketed by the **BX/EX** compatibility operators.

  Non-conformance. `[AdobePDF1.4 3.7.1 12]` allows compatibility operators **BX/EX** to be used.

- `[ISO19005-1:2005 6.2.10 2]` When using the **ri** operator in a content stream, the intent operand shall be on of *RelativaColorimetric*, *AbsoluteColorimetric*, *Perceptual* or *Saturation*.

  Non-conformance. `[AdobePDF1.4 4.5.4]` allows extensions to this set: *Note, however, that the exact set of rendering intents supported may vary from one output device*

to another; a particular device may not support all possible intents, or may support additional ones beyond those listed in the table.

C.3 Fonts

- `[ISO19005-1:2005 6.3.3.1 1]` For any given composite (Type 0) font referenced within a conforming file, the Registry and Ordering strings of the **CIDSystemInfo** entries of **CIDFont** and **CMap** dictionaries for that font shall be identical, unless the value of the CMap dictionary **UserCMap** key is *Identity-H* or *Identity-V*.

  Non-conformance. `[AdobePDF1.4 5.6.2 4]` specifies that this requirement should be met for proper behavior, but it is not required: *For proper behavior, the* **CIDSystemInfo** *entry of a CMap should be compatible with that of the CIDFont or CIDFonts with which it is used. If they are incompatible, the effects produced will be unpredictable.*

- `[ISO19005-1:2005 6.3.3.1 1]` For all Type 2 **CIDFonts**, the CIDFont dictionary shall contain a **CIDToGIDMap** entry that shall be a stream mapping from CIDs to glyph indices or the name *Identity*.

  Non-conformance. `[AdobePDF1.4 5.6.3 3]` specifies that this entry is optional, but not required.

- `[ISO19005-1:2005 6.3.3.1 1]` All CMaps used within a conforming file, except *Identity-H* and *Identity-V*, shall be embedded in that file. For those CMaps that are embedded, the integer value of the **WMode** entry in the CMap dictionary shall be identical to the **WMode** value in the embedded CMap stream.

  Non-conformance. `[AdobePDF1.4 5.6.4]` is not completely aligned with this. Exceptions of embedded CMaps are *for character encodings that are not predefined*, rather than particular *Identity-H* and *Identity-V* CMaps. The **WMode** entry of the CMap diciontary is optional, not required. Additionally, both **WMode** values (CMap dictionary and embedded CMap stream) should (and not shall) be identical.

- `[ISO19005-1:2005 6.3.4 1,2,3,4,5,6]` The font programs for all fonts used within a conforming file shall be embedded within that file, except when the fonts are used exclusively with text rendering mode 3. A font is considered to be used if any of its glyphs are referenced in any of the following contexts: *a)* the **Contents** stream of a page object; *b)* the stream of a Form XObject; *c)* the appearance stream of an annotation, including form fields; *d)* the content stream of a Type 3 font glyph; *e)* the stream of a tiling pattern.

  Non-conformance. `[AdobePDF1.4 5.8]` does not restrict this.

- `[ISO19005-1:2005 6.3.4 7]` Only fonts that are legally embeddable in a file for unlimited, universal rendering shall be used.

  Non-conformance. `[AdobePDF1.4 5.8 2]` does not require an unlimited nor universal rendering for the font program. Additionally, it states that not embeddable fonts

should not (not shall not) be used: *One of the conditions may be that the font program cannot be embedded, in which case it should not be incorporated into a PDF file.*

- `[ISO19005-1:2005 6.3.4 8]` All conforming readers shall use the embedded fonts, rather than other locally resident, substituted or simulated fonts, for rendering.

  Non-conformance. `[AdobePDF1.4 5.8]` does not restrict this.

- `[ISO19005-1:2005 6.3.5 1]` Type 0 **CIDFont** and Type 1 and TrueType font subsets may be used if the embedded font programs define all appropriate glyphs.

  Non-conformance. `[AdobePDF1.4 5.5.3]` does not restrict this.

- `[ISO19005-1:2005 6.3.5 2]` For all Type 1 font subsets referenced within a conforming file, the font descriptor dictionary shall include a **CharSet** string listing the character names defined in the font subset.

  Non-conformance. `[AdobePDF1.4 5.7 3]` says that the **CharSet** entry is optional.

- `[ISO19005-1:2005 6.3.5 3]` For all **CIDFont** subsets referenced within a conforming file, the font descriptor dictionary shall include a **CIDSet** stream identifying which CIDs are present in the embedded **CIDFont** file.

  Non-conformance. `[AdobePDF1.4 5.7.2 7]` says that the **CIDSet** stream is optional.

- `[ISO19005-1:2005 6.3.6 1]` For every font embedded in a conforming file, the glyph width information stored in the **Widths** entry of the font dictionary and in the embedded font program shall be consistent.

  Non-conformance. `[AdobePDF1.4 5.5.1 4]` only requires this for Type 1 fonts, but not for every embedded font in the file.

- `[ISO19005-1:2005 6.3.7 1]` All non-symbolic TrueType fonts shall specify *MacRomanEncoding* or *WinAnsiEncoding* as the value of the **Encoding** entry in the font dictionary. All symbolic TrueType fonts shall not specify an **Encoding** entry in the font dictionary, and their font programs' "cmap" tables shall contain exactly one encoding.

  Non-conformance. `[AdobePDF1.4 5.5.5]` suggests this only as a guideline.

- `[ISO19005-1:2005 6.3.8 2,3,4,5]`[1] The font dictionary shall include a **ToUnicode** entry whose value is a CMap stream object that maps character codes to Unicode values, unless the font meets any of the following three conditions: *a)* fonts that use the predefined encodings **MacRomanEncoding**, **MacExpertEncoding** or **WinAnsiEncoding**, or that use the predefined **Identity-H** or **Identity-V** CMaps; *b)* Type 1 fonts whose character names are taken from the Adobe standard Latin character set or the set of named characters in the Symbol font; *c)* Type 0 fonts whose descendant **CIDFont** uses the **Adobe-GB1**, **Adobe-CNS1**, **Adobe-Japan1** or **Adobe-Korea1**

[1] This requirement is applicable only for files meeting Level A conformance (PDF/A-1a).

character collections.

Non-conformance. `[AdobePDF1.4]` does not restrict this.

C.4 Transparency

- `[ISO19005-1:2005 6.4 1]` If an **SMask** key appears in an **ExtGState** or **XObject** dictionary, its value shall be *None*.

  Non-conformance. `[AdobePDF1.4]` Does not restrict this.

- `[ISO19005-1:2005 6.4 2]` A **Group** object with an **S** key with a value of *Transparency* shall not be included in a form **XObject**.

  Non-conformance. `[AdobePDF1.4]` Does not restrict this.

- `[ISO19005-1:2005 6.4 3,4]` If the **BM** key is present in an **ExtGState** object, its value shall be *Normal* or *Compatible*.

  Non-conformance. `[AdobePDF1.4]` Does not restrict this.

- `[ISO19005-1:2005 6.4 3,5]` If the **CA** key is present in an **ExtGState** object, its value shall be *1.0*.

  Non-conformance. `[AdobePDF1.4]` Does not restrict this.

- `[ISO19005-1:2005 6.4 3,6]` If the **ca** key is present in an **ExtGState** object, its value shall be *1.0*.

  Non-conformance. `[AdobePDF1.4]` does not restrict this.

C.5 Annotations

- `[ISO19005-1:2005 6.5.1 1]` Conforming interactive readers shall provide a mechanism to display the values of the **Contents** key of annotation dictionaries.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.5.2 1]` Annotation types not defined in PDF Reference shall not be permitted. Additionally, the **FileAttachment**, **Sound** and **Movie** types shall not be permitted.

  Non-conformance. `[AdobePDF1.4 8.4.5 1]` states that *plug-in extensions may add new annotation types, and further standard types may be added in the future.* None of these extensions are supported by `[ISO19005-1:2005]`. Additionally, requirement `[AdobePDF1.4 8.4.5 2]` allows the **FileAttachment**, **Sound** and **Movie** annotation types.

- `[ISO19005-1:2005 6.5.3 1]` An annotation dictionary shall not contain the **CA** key with a value other than *1.0*.

  Non-conformance. `[AdobePDF1.4 8.4.1 2]` does not restrict this.

- `[ISO19005-1:2005 6.5.3 2]` An annotation dictionary shall contain the **F** key. The **F** key's **Print** flag bit shall be set to *1* and its **Hidden**, **Invisible** and **NoView** flag bits shall be set to *0*.

  Non-conformance. `[AdobePDF1.4 8.4.1 2]` states that the **F** key is optional. The flag bits values are not restricted.

- `[ISO19005-1:2005 6.5.3 3]` Text annotations should[2] set the **NoZoom** and **NoRotate** flag bits of the **F** key to *1*.

  Non-conformance. `[AdobePDF1.4 8.4.2 2]` is less restrictive regarding this requirement.

- `[ISO19005-1:2005 6.5.3 5]` An annotation dictionary shall not contain the **C** array or the **IC** array unless the colour space of the **DestOutputProfile** in the PDF/A-1 OutputIntent dictionary is *RGB*.

  Non-conformance. `[AdobePDF1.4 8.4.1 2]` states that these entries are optional and do not depend on the colour space of the **DestOutputProfile** in the OutputIntent dictionary.

- `[ISO19005-1:2005 6.5.3 7]` If an annotation dictionary contains the **AP** key, the appearance dictionary that it defines as its value shall contain only the **N** key, whose value shall be a stream defining the appearance of the annotation.

  Non-conformance. `[AdobePDF1.4 8.4.4 11]` Appearance dictionaries may have additional optional keys.

C.6 Actions

- `[ISO19005-1:2005 6.6.1 1]` The **Launch**, **Sound**, **Movie**, **ResetForm**, **ImportData** and **JavaScript** actions shall not be permitted.

  Non-conformance. `[AdobePDF1.4 8.5]` allows all of these actions.

- `[ISO19005-1:2005 6.6.1 1]` Deprecated **set-state** and **no-op** actions shall not be permitted.

[2] This requirement is not a must, but a recommendation. A note explains it: **NoZoom** *and* **NoRotate** *flags are permitted, which allows the use of annotation types that have the same behaviour as the commonly-used text annotation type. By definition, text annotations exhibit the* **NoZoom** *and* **NoRotate** *behaviour even if the flags are not set; explicitly setting these flags removes any potential ambiguity between the annotation dictionary settings and reader behaviour.*

Non-conformance. [AdobePDF1.4 8.5.3 3] states that these actions are obsolete and its use is no longer recommended. However, it does not restrict its use in PDF documents.

- [ISO19005-1:2005 6.6.1 1] Named actions other than **NextPage**, **PrevPage**, **FirstPage**, and **LastPage** shall not be permitted.

  Non-conformance. [AdobePDF1.4 8.5.3] states that additional named actions may be supported by viewer applications: *Viewer applications may support additional, non-standard named actions, but any document using them will not be portable. If the viewer encounters a named action that is inappropriate for a viewing platform, or if the viewer does not recognize the name, it should take no action.*

- [ISO19005-1:2005 6.6.1 2] Interactive form fields shall not perform actions of any type.

  Non-conformance. [AdobePDF1.4 8.6.4] defines and allows the use of form fields actions.

- [ISO19005-1:2005 6.6.2 1] A Widget annotation dictionary or Field dictionary shall not include an **AA** entry for an additional-actions dictionary. The document catalog dictionary shall not include an **AA** entry for an additional-actions dictionary.

  Non-conformance. [AdobePDF1.4] states the the use of the **AA** entry in these three dictionaries is optional.

- [ISO19005-1:2005 6.6.3 1] Conforming interactive readers shall provide a mechanism to display the **F** and **D** keys of a **GoToR** action dictionary, the **URI** key of a **URI** action dictionary, and the **F** key of a **SubmitForm** action dictionary.

  Non-conformance. This is an addition to the rendering behaviour described in specification [AdobePDF1.4].

C.7 Metadata

A PDF document may include general information such as the document's title, author, and creation and modification dates. Such global information about the document itself (as opposed to its content or structure) is called *metadata*, and is intended to assist in cataloguing and searching for documents in external databases. A document's metadata may also be added or changed by users or plug-in extensions.

Metadata is essential for effective management of a file throughout its life cycle. A file depends on metadata for identification and description, as well as for describing appropriate technical and administrative matters. As a result, writers of conforming files may have to comply with various domain-specific metadata requirements defined external to [ISO19005-1:2005].

- [ISO19005-1:2005 6.7.2 1] The document catalog dictionary of a conforming file shall contain the Metadata key. The metadata stream that forms the value of that key

shall conform to XMP Specification. All metadata properties embedded in a file shall be in XMP form except for document information dictionary entries that have no XMP analogues. Properties specified in XMP form shall use the predefined schemas defined in XMP Specification 4 or extension schemas that comply with XMP Specification 4 and requirement `[ISO19005-1:2005 6.7.8]`. Metadata object stream dictionaries shall not contain the Filter key.

Non-conformance. `[AdobePDF1.4 3.6.1 2]` states that the **Metadata** entry is optional.

- `[ISO19005-1:2005 6.7.3]` If a document information dictionary appears within a conforming file, then all of its entries that have analogous properties in predefined XMP schemas shall also be embedded in the file in XMP form. Both values shall be equivalent. These entries are **Title** (*dc:title*), **Author** (*dc:creator*), **Subject** (*dc:subject*), **Keywords** (*pdf:keywords*), **Creator** (*xmp:CreatorTool*), **Producer** (*pdf:Producer*), **CreationDate** (*xmp:CreateDate*) and **ModDate** (*xmp:ModifyDate*). If the *dc:creator* property is present in XMP metadata then it shall be represented by an ordered Text array of length one whose single entry shall consist of one or more names. Equivalence between Author and *dc:creator*, shall be on a character-by-character basis, independent of encoding, comparing the numeric ISO/IEC 10646-1 code points for the characters. Date properties are formatted as a variable-length sequence of temporal components ranging in granularity: year, month, day, hour, minute, second. For properties that map between the PDF date type and the XMP Date type value equivalence shall be on a component-by-component basis, relative to Coordinated Universal Time (UTC).

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.7.4]` All XMP schemas should define the normalization rules that are applicable for their properties. For all metadata properties defined in schemas that do provide normalization rules, the property values shall be entered, saved and retained in the normalized fashion defined by those schemas.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.7.5]` The **bytes** and the **encoding** attributes shall not be used in the header of an XMP packet.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.7.6]` A conforming file should[3] have one or more metadata properties to characterize, categorize and otherwise identify the file (e.g. with *UUID*). Identifiers may be included through use of the **xmp:Identifier** property; use of the **xmpMM:DocumentID**, **xmpMM:VersionID** and **xmpMM:RenditionClass** properties; or use of properties from an extension schema. If a conforming file is changed in any way then the changing identifier part of the file trailer dictionary ID key should be modified accordingly.

[3]This requirement is not a must, but a recommendation.

Non-conformance. [AdobePDF1.4] does not require this.

- [ISO19005-1:2005 6.7.7] In order to describe all high-level user actions taken to create, transform or otherwise instantiate a conforming file, each of those actions should[4] be recorded in the **xmpMM:History** property. For each action that is recorded: *a)* the **action**, **parameters** and **when** fields shall be specified; *b)* the **softwareAgent** field should be specified; *c)* the **instanceID** field shall not be specified.

  Non-conformance. [AdobePDF1.4] does not require this.

- [ISO19005-1:2005 6.7.8] All extension schemas used in a conforming file shall have their descriptions embedded within that file in the metadata stream. These descriptions shall be specified using the PDF/A extension schema description schema defined in [ISO19005-1:2005].

  Non-conformance. [AdobePDF1.4] does not require this.

- [ISO19005-1:2005 6.7.9] All content of all XMP packets shall be well-formed as defined by XML 1.0, 2.1, and RDF/XML Syntax Specification.

  Non-conformance. [AdobePDF1.4] does not require this.

- [ISO19005-1:2005 6.7.10] For all embedded Type 0, Type 1, or TrueType font programs, the embedded font file stream dictionary should[5] include a **Metadata** entry whose value is an XMP metadata stream. The following XMP metadata elements should be supplied: **xmp:Title**, giving the value of the **FontName** key from the font's font descriptor dictionary; **xmpRights:Copyright**, giving the copyright statement; **xmpRights:Marked**, with the Boolean value *true*; **xmpRights:Owner**, giving the legal owner of the font; **xmpRights:UsageTerms**, giving a statement of the licensing terms under which the font is being used.

  Non-conformance. [AdobePDF1.4] does not require this.

- [ISO19005-1:2005 6.7.11] The PDF/A version and conformance level of a file shall be specified using the PDF/A Identification extension schema defined in requirement [ISO19005-1:2005 6.7.11].

  Non-conformance. [AdobePDF1.4] does not require this.

C.8 Logical structure

Requirements in this subsection are applicable only for files meeting PDF/A Level A conformance. For Level B conformance these requirements can be ignored.

The intent of the requirements in this subsection is to ensure the recovery of the textual content of a conforming file as a sequence of words defined in the natural reading order of

[4] This requirement is not a must, but a recommendation.

[5] This requirement is not a must, but a recommendation.

the language in which they are written. Similarly, it ensures that the individual characters of each word are recoverable in their natural reading order. Furthermore, these requirements allow the recovery of higher-level semantic information concerning the logical structure of the document.

- `[ISO19005-1:2005 6.8.2.1 1]` A Level A conforming file shall meet all of the requirements set forth for Tagged PDF in `[AdobePDF1.4]`.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.2.2 1]` The document catalog dictionary shall include a **MarkInfo** dictionary whose sole entry, **Marked**, shall have a value of *true*.

  Non-conformance. `[AdobePDF1.4 9.7.1 2]` states that this entry is optional, but not mandatory. Its value may be *true* or *false*.

- `[ISO19005-1:2005 6.8.3.1 1]` Pagination features such as running heads or page numbers, cosmetic layout features such as footnote rules or background screens, and production aids such as cut marks and colour bars should[6] be specified as pagination, layout, and page artifacts, respectively.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.3.2 1,2]` For languages and script systems that normally use space characters to indicate word breaks, the following additional restriction shall apply: within show strings, word breaks shall be explicitly indicated by the presence of one or more space characters between all of the individual words in the show string. If a word ends at a show string boundary, one or more space characters shall be inserted at the end of the show string. Note that a single word may span two or more show strings; word breaks are indicated only by the explicit presence of one or more space characters, not by the boundaries of a show string. For the purposes of indicating word breaks, a sequence of two or more consecutive space characters is semantically equivalent to a single spacing character.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.3.3 1,2]` The logical structure of the conforming file shall be described by a structure hierarchy rooted in the **StructTreeRoot** entry of the document catalog dictionary. Each structure element dictionary in the structure hierarchy shall have a **Type** entry with the name value of **StructElem**.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.3.4 1,2]` The definition of block-level structuring elements should[7] follow the strongly structured paradigm in `[AdobePDF1.4 9.7.4]`. All non-standard structure types shall be mapped to the nearest functionally equivalent standard type, in the role map dictionary of the structure tree root. This mapping may

[6] This requirement is not a must, but a recommendation.

[7] This requirement is not a must, but a recommendation.

be indirect; within the role map a non-standard type can map directly to another non-standard type, but eventually the mapping must terminate at a standard type.

Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.4 1,2]` The default natural language for all text in a file should[8] be specified by the **Lang** entry in the document catalog dictionary. All textual content within a file which differs from the default language should be indicated by use of a **Lang** property attached to a marked-content sequence, or by a **Lang** entry in a structure element dictionary.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.4 3]` If the **Lang** entry is present in the document catalog dictionary or in a structure element dictionary or property list, its value shall be a language identifier.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.4 4]` All text strings encoded in Unicode whose language is not the default natural language for the file or not the natural language defined by the innermost enclosing structure element or marked-content sequence should[9] indicate their language using the internal escape sequence.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.5 1]` All structure elements whose content does not have a natural predetermined textual analogue should[10] supply an alternate text description using the **Alt** entry in the structure element dictionary.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.6 1]` For annotation types that do not display text, the **Contents** key of an annotation dictionary should[11] be specified with an alternative description of the annotation's contents in human-readable form.

  Non-conformance. `[AdobePDF1.4]` does not require this.

- `[ISO19005-1:2005 6.8.7 1]` All textual structure elements that are represented in a non-standard manner should[12] supply replacement text using the **ActualText** entry in the structure element dictionary.

  Non-conformance. `[AdobePDF1.4]` does not require this.

[8] This requirement is not a must, but a recommendation.
[9] This requirement is not a must, but a recommendation.
[10] This requirement is not a must, but a recommendation.
[11] This requirement is not a must, but a recommendation.
[12] This requirement is not a must, but a recommendation.

- `[ISO19005-1:2005 6.8.8 1]` All instances of abbreviations and acronyms in textual content should[13] be placed in a marked-content sequence with a **Span** tag whose **E** property provides a textual expansion of the abbreviation or acronym.

  Non-conformance. `[AdobePDF1.4]` does not require this.

C.9 Interactive Forms

- `[ISO19005-1:2005 6.9 2]` A conforming reader shall not use form fields to change the rendered representation of the page or the content of the file at any time. A Widget annotation dictionary or Field dictionary shall not contain the **A** or **AA** keys.

  Non-conformance. `[AdobePDF1.4 8.6.2 2]` states that these entries are all optional.

- `[ISO19005-1:2005 6.9 3]` The **NeedAppearances** flag of the interactive form dictionary shall either not be present or shall be *false*.

  Non-conformance. `[AdobePDF1.4 8.6.2 2]` states that this entry is optional, and its values may be both *true* or *false*.

- `[ISO19005-1:2005 6.9 4]` Every form field shall have an appearance dictionary associated with the field's data. A conforming reader shall render the field according to the appearance dictionary without regard to the form data.

  Non-conformance. `[AdobePDF1.4 8.6]` does not restrict this.

[13] This requirement is not a must, but a recommendation.

Appendix D

ANTLR prototypes

This appendix shows the corresponding ANTLR parser generator input in EBNF for the grammars G_1 and G_2.

D.1 EBNF grammars

D.1.1 G_1

```
#lexclass START
#token PDF_TOKEN_TRUE "true"
#token PDF_TOKEN_FALSE "false"
#token PDF_TOKEN_STREAM "stream"
#token PDF_TOKEN_ENDSTREAM "endstream"
#token PDF_TOKEN_NULL "null"
#token PDF_TOKEN_OBJ "obj"
#token PDF_TOKEN_ENDOBJ "endobj"
#token PDF_TOKEN_R "R"
#token PDF_TOKEN_OPERATOR "[a-zA-Z][a-zA-Z]"
#token PDF_TOKEN_INTEGER "[0-9]+"
#token PDF_TOKEN_REAL "([0-9]+\.[0-9]*)|(\.[0-9]+)"
#token PDF_TOKEN_STRING "\(([a-zA-Z0-9\ \-\,\!])*\)|
                         <([a-zA-Z0-9][a-zA-Z0-9])*>"
#token PDF_TOKEN_NAME "/[a-zA-Z0-9\-\.\*]*"
#token PDF_TOKEN_COMMENT "%[a-zA-Z0-9\ \-]*" << zzskip();>>
#token PDF_TOKEN_DICT_START "\<\<"
#token PDF_TOKEN_DICT_END "\>\>"
#token PDF_TOKEN_ARRAY_START "\["
#token PDF_TOKEN_ARRAY_END "\]"
#token PDF_TOKEN_PROC_START "\{"
#token PDF_TOKEN_PROC_END "\}"
#token SPACE "[\ \n\t]" << zzskip();>>

pdf_objects
    : (indirect_object)*
    ;
```

```
indirect_object
    : PDF_TOKEN_INTEGER PDF_TOKEN_INTEGER PDF_TOKEN_OBJ^
    (
            ( PDF_TOKEN_TRUE | PDF_TOKEN_FALSE)
        | PDF_TOKEN_INTEGER ( PDF_TOKEN_INTEGER PDF_TOKEN_R^ | )
        | PDF TOKEN_REAL
        | PDF_TOKEN_STRING
        | PDF_TOKEN_NAME
        | PDF_TOKEN_NULL
        | array_object
        | dictionary_object ( stream_object | )
    ) PDF_TOKEN_ENDOBJ!
    ;

array_object
    : PDF_TOKEN_ARRAY_START^ ( array_or_dictionary_value )*
      PDF_TOKEN_ARRAY_END!
    ;

array_or_dictionary_value
    : ( PDF_TOKEN_TRUE^ | PDF_TOKEN_FALSE^ )
    | PDF_TOKEN_INTEGER ( PDF_TOKEN_INTEGER PDF_TOKEN_R^ | )
    | PDF_TOKEN_REAL
    | PDF_TOKEN_STRING^
    | PDF_TOKEN_NAME^
    | PDF_TOKEN_NULL^
    | array_object
    | dictionary_object
    ;

dictionary_object
    : PDF_TOKEN_DICT_START^ ( key_value_pair )* PDF_TOKEN_DICT_END!
    ;

key_value_pair
    : PDF_TOKEN_NAME^ ( array_or_dictionary_value )
    ;

stream_object
    : PDF_TOKEN_STREAM^ ( PDF_TOKEN_OPERATOR )* PDF_TOKEN_ENDSTREAM! ;
```

D.1.2 G_2

```
#lexclass START
#token PDF_TOKEN_TRUE "true"
#token PDF_TOKEN_FALSE "false"
```

```
#token PDF_TOKEN_STREAM "stream"
#token PDF_TOKEN_ENDSTREAM "endstream"
#token PDF_TOKEN_NULL "null"
#token PDF_TOKEN_OBJ "obj"
#token PDF_TOKEN_ENDOBJ "endobj"
#token PDF_TOKEN_R "R"
#token PDF_TOKEN_OPERATOR "[a-zA-Z][a-zA-Z]"
#token PDF_TOKEN_INTEGER "[0-9]+"
#token PDF_TOKEN_REAL "([0-9]+\.[0-9]*)|(\.[0-9]+)"
#token PDF_TOKEN_STRING "\(([a-zA-Z0-9\ \-\,\!])*\)|
                         <([a-zA-Z0-9][a-zA-Z0-9])*>"
#token PDF_TOKEN_NAME "/[a-zA-Z0-9\-\.\*]*"
#token PDF_TOKEN_COMMENT "%[a-zA-Z0-9\ \-]*" << zzskip();>>
#token PDF_TOKEN_DICT_START "\<\<"
#token PDF_TOKEN_DICT_END "\>\>"
#token PDF_TOKEN_ARRAY_START "\["
#token PDF_TOKEN_ARRAY_END "\]"
#token PDF_TOKEN_PROC_START "\{"
#token PDF_TOKEN_PROC_END "\}"
#token SPACE "[\ \n\t]" << zzskip();>>

pdf_objects
    : (indirect_object)*
    ;

indirect_object
    : PDF_TOKEN_INTEGER PDF_TOKEN_INTEGER PDF_TOKEN_OBJ^ ( contained_object )
      PDF_TOKEN_ENDOBJ!
    ;

contained_object
    : atomic_object
    | array_object
    | dictionary_object ( stream_object | )
    ;

atomic_object
    : ( PDF_TOKEN_TRUE | PDF_TOKEN_FALSE )
    | PDF_TOKEN_INTEGER ( PDF_TOKEN_INTEGER PDF_TOKEN_R^ | )
    | PDF_TOKEN_REAL
    | PDF_TOKEN_STRING
    | PDF_TOKEN_NAME
    | PDF_TOKEN_NULL
    ;

array_object
    : PDF_TOKEN_ARRAY_START^ ( array_or_dictionary_value )*
```

```
      PDF_TOKEN_ARRAY_END
    ;

array_or_dictionary_value
    : atomic_object
    | array_object
    | dictionary_object
    ;

dictionary_object
    : PDF_TOKEN_DICT_START^ ( key_value_pair )* PDF_TOKEN_DICT_END!
    ;

key_value_pair
    : PDF_TOKEN_NAME^ ( array_or_dictionary_value )
    ;

stream_object
    : PDF_TOKEN_STREAM^ ( PDF_TOKEN_OPERATOR )* PDF_TOKEN_ENDSTREAM! ;
```