2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Automated PDF Reporting in MSV Workflow

Barabanov Kostiantyn, CSS ICW, Infineon, Graz, Austria (*Kostiantyn.Barabanov2@infineon.com*)

Suessmuth Clemens, CSS ICW, Infineon, Graz, Austria (*Clemens.Suessmuth@infineon.com*)

***Abstract*—In high-throughput verification environments, timely and accurate reporting is essential for transparency, debugging, and traceability. This paper introduces an automated PDF reporting workflow integrated within an MSV system that revolutionizes the generation of verification reports. By leveraging a combination of Perl and Python scripts, our approach dynamically gathers critical system information, simulation logs, and waveform data to produce a polished, multi-page PDF report. The process seamlessly integrates environment validation, metadata extraction via LDAP queries, and the conversion of simulation outputs into standardized PDF formats. Advanced PDF manipulation techniques—including automated page merging, image embedding, and orientation adjustments—allowed the final report to meet rigorous quality and consistency standards. The resulting framework not only minimizes manual intervention but also enhances reproducibility and traceability in test case analysis. Experimental results demonstrate the system's robustness, scalability, and potential for adoption in other verification domains, paving the way for fully automated, real-time reporting in complex design environments.**

Keywords— AMS, eps, pdf, report, Perl, Python

I. INTRODUCTION

In modern mixed signal simulation (MSV) environments, the complexities of verifying both digital and analog components demand timely, accurate, and comprehensive reporting. Traditionally, compiling detailed reports from diverse simulation outputs, log files, and waveform data has been a manual, error-prone process that fails to keep pace with the rapid development cycles of complex systems. The challenges are further amplified by the need to seamlessly merge data from various sources while preserving the integrity and traceability of test results.

This paper introduces an automated PDF reporting workflow specifically designed for MSV systems. Our solution brings together a combination of Perl and Python scripts to create a unified reporting engine that automates the entire report production process.

II. MOTIVATION

The motivation for implementing automated PDF reporting in MSV workflows is driven by the need to enhance efficiency, improve data accuracy, traceability and support collaborative diagnostics. This automation not only streamlines the verification process but also plays a crucial role in advancing the reliability and pace of mixed signal simulation, ultimately contributing to faster and more robust product development cycles. Key drivers for automation include:

A. Efficiency and Timeliness

Automating the generation of PDF reports drastically reduces the turnaround time between simulation execution and result analysis. Engineers can receive near real-time documentation of test results, allowing for faster identification of issues and more rapid iterations in the design process.

B. Consistency and Accuracy

Manual compilation can lead to errors and incomplete documentation. An automated workflow combines simulation outputs, warnings, and logs into a standardized PDF, ensuring consistent and reproducible reports.

C. Traceability and Documentation

Automated PDF reports embed essential metadata—ranging from simulation parameters to developer information—into each document. This promotes a higher level of traceability, as reports can be directly linked back to specific test cases and simulation conditions. Such comprehensive documentation is invaluable for audits, debugging, and adhering to regulatory or quality assurance standards.

III. OVERVEIW

The block diagram “Figure 1” illustrates a three-step process that transforms simulation results and log outputs into a unified, professional PDF report within the MSV environment.

- **Initialization and Variable Definition.** This initial phase establishes the foundation for the entire reporting workflow by configuring the working environment and defining all necessary variables.
- **Data Conversion and Content Generation.** This phase focuses on creating and converting various components of the final report, including the generation of the title page.
- **Merging into the Final Report.** The final phase focuses on consolidating all the previously generated PDF components into a single, cohesive document.

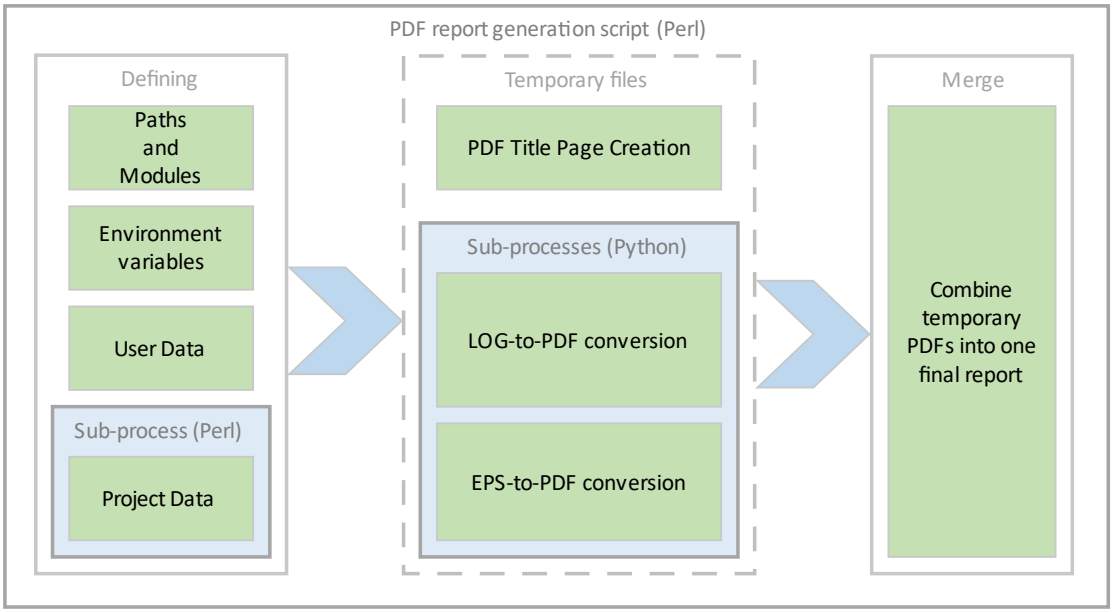


Figure 1. Block diagram of automated PDF report generation workflow

IV. IMPLEMENTATION

A. *PDF report generation script*

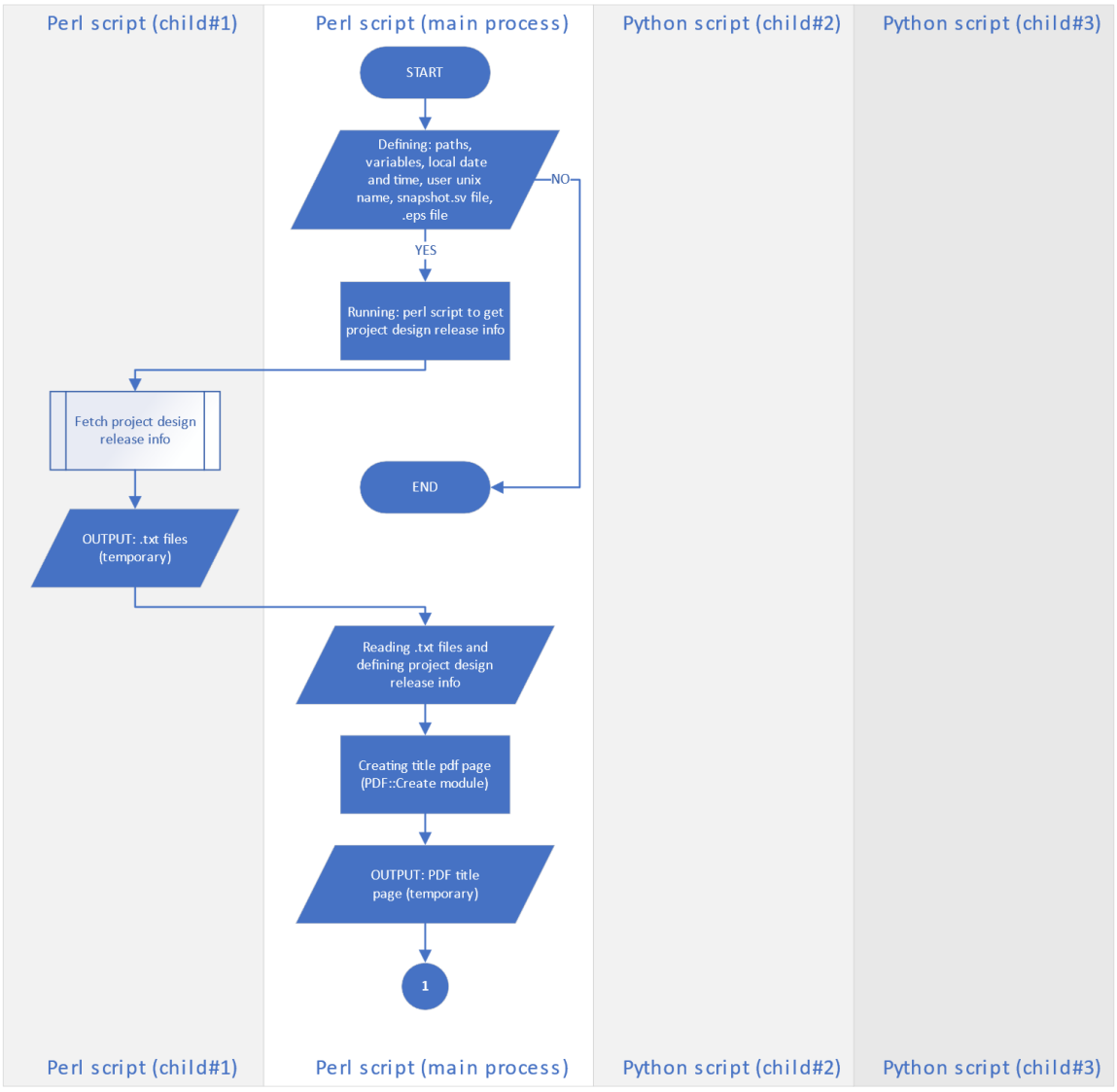
The “Figure 2” illustrates the flowchart of the main PDF report generation script. The script initiates three internal components as sub-processes. The first is a Perl-based script responsible for fetching project design release information. As a child process, it does not share environment variables with the parent process. Therefore, the retrieved release information is written to temporary text files, which are subsequently read by the main script. The second sub-process is a Python-based script dedicated to converting log or text files into PDF format. Details are provided in the “LOG-to-PDF conversion script” section and further elaborated on page 6. The third sub-process is also implemented in Python and is responsible for converting EPS files into PDF format. The algorithm is shown in “EPS-to-PDF conversion script” section, along with relevant implementation details and usage examples.

At the outset, the script dynamically configures its runtime environment by appending local module paths using the DMWA environment variable. It imports key Perl modules—such as PDF::Create [1] for generating PDF content, PDF::API2 [2] for advanced PDF manipulation, and standard libraries for file and path management. In addition, the script retrieves the current working directory and parses it to extract project-relevant information (e.g., test bench and test case names). Crucially, it performs an LDAP query (via a system call to ldaplist [3]) to extract user metadata—including full name, email, and department—which is later used in the document’s title page. This careful preparation (including validation of folder structures, file paths, and naming conventions) lays the groundwork for a reproducible and context-aware report generation process.

The script verifies that the most recent project design release information is available. To obtain this data, it launches a dedicated sub-process (child #1, see "Figure 2"), which gathers the necessary details and writes them to temporary text files. These files are then read by the main script, and the extracted values are stored in variables that populate key fields on the report's title page.

In the next stage, the script constructs the PDF report's title page using the PDF::Create module (see example in "Figure 5"). It initializes a new PDF document. The layout is carefully configured to A4 landscape dimensions, and multiple visual elements are incorporated: high-resolution images (e.g., company logos and headers) and formatted text strings. These textual elements include:

- Project name: "XYZ8"
- The report title: "AMS Verification Report"
- Design configuration details
- TB/TC identifiers
- Author information, metadata
- Creation date, timestamp



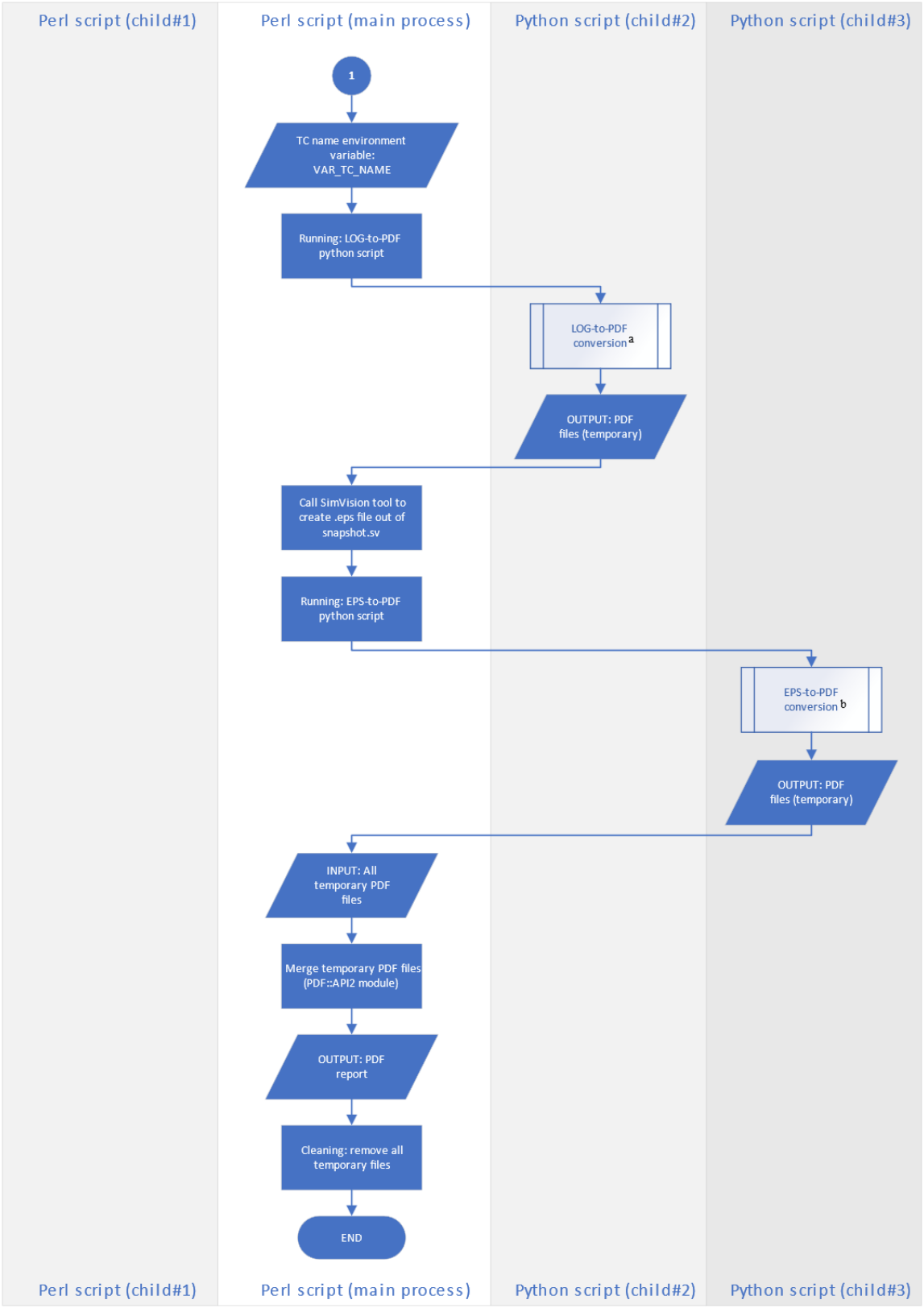


Figure 2. Flowchart of the PDF report generation script

[a.] *Described on page 5 LOG-to-PDF conversion script*

[b.] *Described on page 7 EPS-to-PDF conversion script*

Following the title page generation, the script proceeds to convert simulation artifacts into PDF format. A Python script (child #2) is invoked to transform test logs into a visually consistent PDF. To handle waveform data, the script runs SimVision tool to open pre-saved "snapshot.sv" file and export the relevant waveform view as an EPS image. Once generated, the EPS file is passed to a third subprocess (child #3), which converts it to PDF. Finally, the script post-processes the waveform pages using the PDF::API2 module, rotating them 90° for proper orientation. These steps ensure both numerical logs and waveform snapshots are encapsulated in clean, standardized documents.

In the final production stage, the script merges several PDF files—comprising the title page, log report, measurement outputs, and waveform document—into a singular, cohesive PDF report. The merging is accomplished via iterative importation of individual PDF pages using the PDF::API2 module, with subsequent deletion of intermediate files to maintain a clean workspace. To facilitate broad distribution, the merged report is then copied to predetermined locations, including documentation directories and the user's desktop environment. This deployment step underscores the system's orientation toward seamless integration into broader design and test pipelines.

As a concluding operation, the script performs a cleanup process by recursively deleting all temporary files from its working directory. This ensures that the runtime environment is reset for subsequent invocations and integrates with larger automated workflows where temporary state must not persist between runs.

The "Table I" outlines the key processing steps of the PDF report generation script, categorizing each component by its role—either as input or output—and providing a concise description of its function within the overall PDF report generation workflow. It captures how data flows through the system: from gathering user and environment metadata, through file conversions and PDF assembly, to final cleanup and export.

Table I. Input/Output and processing of PDF report generation script

Component	Type	Description
Module & lib imports	Input	Load Perl modules and custom libs (PDF::Create, PDF::API2, etc.)
Environment & path setup	Input	Read $ENV{DMWA}, cwd, build bin/doc/pdf/tmp paths, capture PID, date/time, TB/TC names
Perl script (child#1 sub-process)	Input (temporary .txt files)	Extract project configuration data from temporary text files and assign them to variables in the parent Perl script.
PDF:Create module	Process (module call)	Create an A4-landscape PDF, embed the header and logo, writes all metadata (config. versions, TB/TC names, user info, date/time)
Title-page PDF	Output (temporary .pdf file)	Save the title page as temporary PDF file
VAR_TC_NAME	Output (environment variable)	TC name variable to be visible for all child processes
Python script (child#2 sub-process)	Input (temporary .pdf files)	Converts .log/.txt simulation files into a formatted PDF format
SimVision tool	Process (invoke external tool)	Runs SimVision tool to open pre-saved "snapshot.sv" file and export the relevant waveform view as an EPS image.
Python script (child#3 sub-process)	Input (temporary .pdf file)	Converts .eps waveform file into a formatted PDF format
PDF report	Output	Combines temporary PDFs into one final report
Copy & cleanup	Process (file operation)	Copies final report to Desktop/docs, removes all temporary files

B. LOG-to-PDF conversion script

The flowchart "Figure 3" illustrates the sequential logic employed by the Python-based script that automates the conversion of .log and .txt files into formatted .pdf documents. All processes occur in a linear, unconditional sequence, with no branching logic or parallel execution paths. This makes the script suitable for deployment in controlled automation pipelines or continuous integration systems.

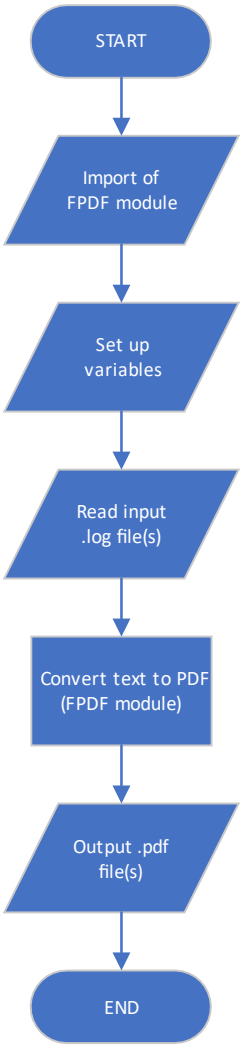


Figure 3. Flowchart of the LOG-to-PDF conversion script

The “Table II” presents a concise summary of the input sources, processing components, and output artifacts involved in the log-to-pdf conversion script. It identifies key environmental variables, file dependencies, and functional steps, along with their respective roles in the data transformation pipeline.

Table II. Input/Output and processing summary of log-to-pdf conversion script

Component	Type	Description
FPDF	Input	FPDF module. Provides the functionality to set up the PDF layout, add pages, set fonts, and write text to the PDF
textwrap	Input	Python’s built-in module. Provides functions for wrapping and formatting text
DMWA	Input (environment variable)	Path to local Python modules (FPDF)
VAR_TC_NAME	Input (environment variable)	TC name variable passed from a Perl script (parent process) to construct input_filename_1 and output_filename_1 files.
input_filename_1	Input (.log or .txt file)	Primary log file to convert to PDF
input_filename_2	Input (.log or .txt file)	Secondary log file to convert to PDF
text_to_pdf()	Process (invoke function)	Converts text to a PDF document using FPDF module

Component	Type	Description
output_filename_1	Output (.pdf file)	PDF file created from input_filename_1
output_filename_2	Output (.pdf file)	PDF file created from input_filename_2
Console print messages	Output (console)	Terminal messages indicating file conversion status

The FPDF [4], [5] Python library was selected due to its lightweight footprint, platform independence, and native PDF generation capabilities without requiring external dependencies. Unlike PDF solutions that rely on LaTeX, report engines, or graphical user interfaces (GUI), FPDF allows fully programmatic, headless operation—ideal for integration in automated workflows and CI environments.

Key advantages include:

- Precise layout control: Enables customized font sizes, orientations, margins, and page breaks—essential for rendering structured log content onto A4 landscape pages.
- Monospaced font support: Preserves alignment in log output with clear readability.
- No installation overhead: Modules can be locally bundled and imported from a script-defined path, ensuring compatibility in constrained environments without system-wide installs.

Given the need for portability, minimal system configuration, and reliable document output, fpdf was a natural choice for the logging pipeline.

C. EPS-to-PDF conversion script

The flowchart “Figure 4” illustrates the logical structure and procedural flow of a Python-based script designed to convert Encapsulated PostScript (.eps) files into Portable Document Format (.pdf) using the epstopdf utility [6].

The process begins with provision of an input argument—specifically, the path to an EPS file—supplied via the command line. The script then checks whether the correct number of arguments was passed. If input is incorrect, the program outputs usage guidance and exits gracefully with a non-zero code.

Upon successful input validation, the core function eps_to_pdf() is invoked. It orchestrates the conversion pipeline by constructing the output filename, formulating the system-level command, and executing the transformation via a shell call to epstopdf. The function then returns the path to the generated PDF.

An optional but recommended decision node checks the exit status of the conversion to ensure reliability, enabling appropriate feedback or error reporting. If the command executes successfully, a confirmation message is printed to the user. The process then concludes.

The “Table III” Input/Output and processing summary of eps-to-pdf conversion script provides a structured overview of the script’s key components, outlining the required input, resulting output, core processing steps, and associated system interactions involved in converting .eps files to .pdf format using the epstopdf external utility within a Python environment.

Table III. Input/Output and processing summary of eps-to-pdf conversion script

Component	Type	Description
input_file	Input (CLI Arg)	Path to the .eps file specified as a command-line interface (CLI) argument
subprocess.call()	Process (command execution)	Executes the epstopdf command using the constructed input/output file paths
epstopdf	Process (invoke external tool)	Shell-invoked utility that performs the EPS-to-PDF conversion
output_file	Output (.pdf file)	Generated .pdf file with the same base name as the input

Component	Type	Description
stdout message	Output (console)	Confirmation message printed upon successful conversion

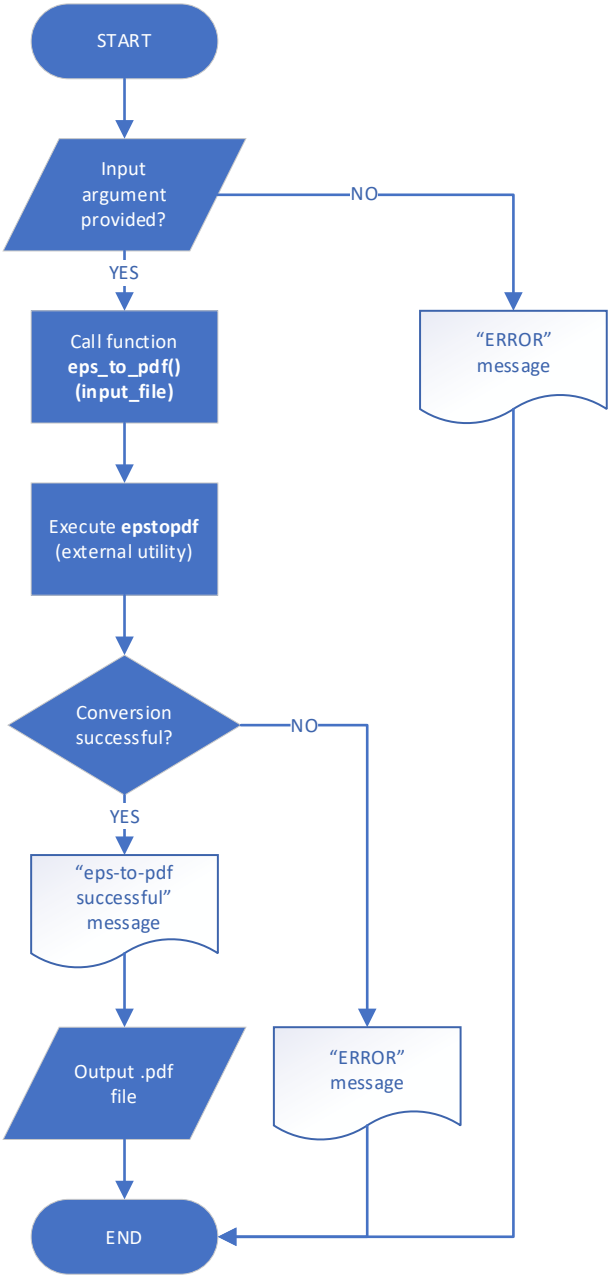


Figure 4. Flowchart of the EPS-to-PDF conversion script

The development process began with an investigation into various methods for automating the conversion of Encapsulated PostScript (EPS) files to PDF within a Unix-based Python environment. Several options were evaluated, including Perl modules, Python-native libraries, and commercial APIs. However, many of these introduced unnecessary dependencies, lacked full support for vector fidelity, or required complex installations that were unsuitable for lightweight scripting.

During this evaluation, community best practices highlighted on platforms such as TeX Stack Exchange [6] consistently recommended the use of the epstopdf utility for high-quality, scriptable EPS-to-PDF conversion. Based

on these insights and the detailed functionality described in the official documentation by H. Oberdiek [7], the epstopdf command-line tool was selected for its reliability and compatibility with Unix systems via TeX Live.

Notably, epstopdf is also deeply integrated into LaTeX workflows, where it is invoked automatically to convert EPS graphics for inclusion in documents compiled with PDFLaTeX [8]. This seamless integration ensures that authors can use vector graphics without manual conversion steps, maintaining both document fidelity and workflow efficiency.

By incorporating this utility into a Python script through the subprocess module, the resulting solution remains simple, portable, and robust—making it well-suited for automated scientific and document production workflows, and fully compatible with established LaTeX practices.

V. CHALLENGES

A. *Environment and Directory Validation*

Dynamic Environment Variables: Maintaining consistency in environment variable settings (e.g., $DMWA) across diverse platforms was critical. The solution required robust checks to ensure that all necessary directories (e.g., bin, doc, lib, script folder and temporary paths) were correctly defined and accessible.

Folder Structure Requirements: Scripts were designed to run from specific test case directories (e.g., tc_*), making dynamic validation of the execution context essential. Non-compliant setups required thoughtful error messaging and controlled exits to ensure reliable workflow behavior.

B. *Dependency and Module Management*

Heterogeneous Toolchain: The system integrates multiple Perl modules (e.g., PDF::Create, PDF::API2, File) and Python scripts for file conversion and PDF manipulation. Managing different module versions and ensuring compatibility across third-party libraries necessitated thorough documentation and testing.

Local Module Modifications: Without admin rights, custom changes to modules like PDF::Font.pm were necessary. Ensuring compatibility and avoiding disruption to module interoperability posed a key challenge.

C. *Data Extraction and Parsing*

User Metadata Retrieval: The reporting workflow extracts user information via LDAP queries and text parsing operations. Handling diverse output formats and taking care that whitespace and delimiters are correctly managed was non-trivial, especially in heterogeneous network environments, “Figure 5”.

Log and Snapshot File Processing: Parsing log files “Figure 6” and simulation outputs “Figure 7” demanded strong string handling and error checks. Missing or malformed files (such as waveform or snapshots in .sv or .eps formats “Figure 8”) required extra validation to prevent report errors.

D. *Interoperability Between Perl and Python*

Cross-Language Integration: Coordinating tasks between Perl and Python (e.g., converting EPS to PDF) involved complexities in environment variable propagation and exit-status monitoring. Preventing errors in one language module from halting the entire workflow required careful synchronization and error-handling strategies.

System Call Reliability: The use of system calls to launch external scripts and applications (e.g., simvision and Python-based converters) introduced potential race conditions and dependency on the underlying shell environment. This demanded rigorous testing and fallback mechanisms to provide robustness.

E. *File Manipulation and Cleanup*

Temporary File Handling: The workflow generates several temporary files during execution, including intermediate PDF pages and log files. It was crucial to properly clean up these files, particularly in the event of failures, to prevent clutter and maintain deterministic results across repeated runs.

PDF Merging and Format Consistency: Merging multiple PDF components, each constructed under varying conditions (i.e., varying page orientations, different source resolutions), required precise control over formatting parameters. Handling discrepancies arising from different PDF modules and providing a consistent final output was a detailed and iterative process.

VI. RESULTS

In conclusion, the newly developed automated PDF reporting workflow has proven to be a highly useful and efficient solution for streamlining the generation of verification reports in MSV systems. This innovative approach has been successfully applied in a real project, demonstrating its practical value and effectiveness. The workflow's push-button solution enables users to effortlessly generate comprehensive and standardized reports, significantly boosting productivity and reducing manual effort. By automating the reporting process, users can focus on higher-level tasks, such as analysis and decision-making, rather than tedious and time-consuming report generation.

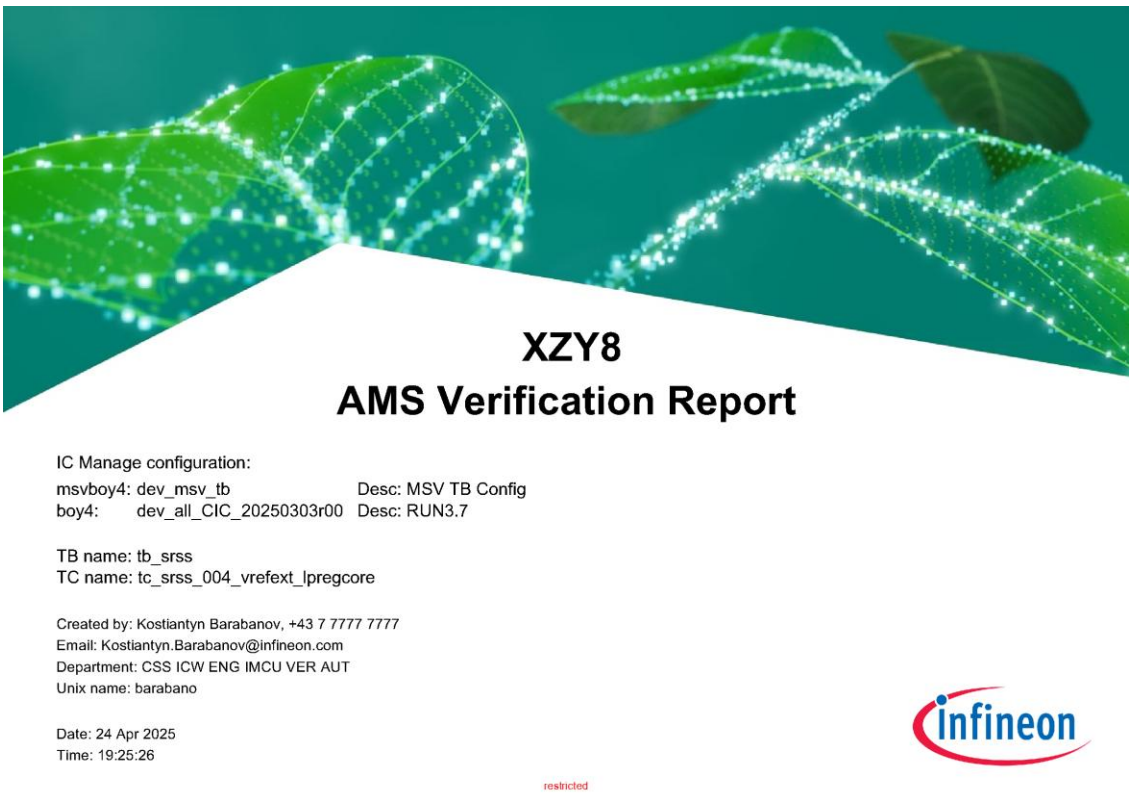


Figure 5. Title page of pdf report

```
SUMMARY OF WARNINGs/ERRORs PRESENT IN THE LOG FILES:

Path: /home/xyz8.ivcs/barabano/barabano_boy4_1.0-dev_56/msvboy4/tb/lab_msv_boy4/tb_srss/tc_srss_004_vrefext_lpregcore/tc_srss_004_vrefext_lpregcore.log
Date: 24 Apr 2025
Time: 19:22:26

#LOG1# /home/xyz8.ivcs/barabano/barabano_boy4_1.0-dev_56/msvboy4/tb/lab_msv_boy4/tb_srss/tc_srss_004_vrefext_lpregcore/rundir/meas_output.log
   || All measured values are PASSED!

#LOG2# /home/xyz8.ivcs/barabano/barabano_boy4_1.0-dev_56/msvboy4/tb/lab_msv_boy4/tb_srss/tc_srss_004_vrefext_lpregcore/rundir/measures.log
   || No Warnings found!
   || No Errors found!

#LOG3# /home/xyz8.ivcs/barabano/barabano_boy4_1.0-dev_56/msvboy4/tb/lab_msv_boy4/tb_srss/tc_srss_004_vrefext_lpregcore/rundir/meas_output_aux.log
*X || meas_output_aux.log is not EXIST!

#LOG4# /home/xyz8.ivcs/barabano/barabano_boy4_1.0-dev_56/msvboy4/tb/lab_msv_boy4/tb_srss/tc_srss_004_vrefext_lpregcore/rundir/measures_aux.log
*X || measures_aux.log is not EXIST!

#LOG5# /home/xyz8.ivcs/barabano/barabano_boy4_1.0-dev_56/msvboy4/tb/lab_msv_boy4/tb_srss/tc_srss_004_vrefext_lpregcore/rundir/irun.log
*W || 272 Warnings are present, please check the log file.
*E || 26 Errors are present, please check the log file.

#LOG6# /opt/tmp_share/ifxcy/barabano/barabano_boy4_1.0-dev_boy4_soc_tb/vboy4/tb/fnv/run_dgs/tests/msv/tc_srss_004_vrefext_lpregcore__inst0_syscpuss_core0_s/base_test/base_test_RTL.log
*W || 77 Warnings are present, please check the log file.
   || No Errors found!

---------------
TC STATUS: PASS!
```

Figure 6. The second pdf report page - comprehensive log files analysis

POST-SIMULATION OCEAN REPORT:

Path: /home/xyz8.ivcs/barabano/barabano_boy4_1.0-dev_56/msvboy4/tb/lab_msv_boy4/tb_srss/tc_srss_004_vrefext_lpregcore/rundir/meas_output.log
Name: barabano
Date: 24 Apr 2025
Time: 19:22:26

No	Type	Var. Name	Meas. value	Expected val.	Spec. Min.	Spec. Max.	P/F	Description
0	[---]	tcr	1.43624e-03	AVAILABLE			PASS	cross time rise edge of 'pwr_adft_select_qq[31]' signal of 'u_srss_tdep_regs' block
0	[---]	tx1	8.00000e-08	AVAILABLE			PASS	+ time
0	[---]	tx2	1.00000e-03	AVAILABLE			PASS	+++ time
0	[---]	tm1	1.43632e-03	AVAILABLE			PASS	measure time = cross time 'tcr' + time 'tx1'
0	[---]	tm2	2.43624e-03	AVAILABLE			PASS	measure time = cross time 'tcr' + time 'tx2'
1	[REG]	r1_adft_sel_h	a0000000	a0000000			PASS	'pwr_adft_select_qq[31:0]' register measured @ tm1
2	[REG]	r2_adft_sel_h	a0310000	a0310000			PASS	'pwr_adft_select_qq[31:0]' register measured @ tm2
3	[ANA]	v_vrefext	8.300e-01		7.470e-01	9.130e-01	PASS	'vrefext' signal of 'u_s40srssa_lpregcore' block measured @ tm2
4	[ANA]	v_gpio_p2_0	8.300e-01		7.470e-01	9.130e-01	PASS	GPIO 'pad' signal of 's40iolib_gpio_ovt_p2_0_0' block measured @ tm2
5	[ANA]	v_p2_0_pad	8.300e-01		7.470e-01	9.130e-01	PASS	PAD 'p2_0_pad' signal of 'DUT' block measured @ tm2
6	[PWR]	vcc*	1.096e+00		1.089e+00	1.111e+00	PASS	measure LV 'vccact_in' voltage value of 'u_s40srssa_coreana' block @ tm2
7	[PWR]	vcc*	1.096e+00		1.089e+00	1.111e+00	PASS	measure LV 'vnwell_in' voltage value of 'u_s40srssa_coreana' block @ tm2
8	[PWR]	vcc*	1.096e+00		1.089e+00	1.111e+00	PASS	measure LV 'vccret_in' voltage value of 'u_s40srssa_coreana' block @ tm2
9	[PWR]	vcc*	1.096e+00		1.089e+00	1.111e+00	PASS	measure LV 'vccdpslp' voltage value of 'u_s40srssa_coreana' block @ tm2
10	[PWR]	vcc*	1.096e+00		1.089e+00	1.111e+00	PASS	measure LV 'vccd_in ' voltage value of 'u_s40srssa_coreana' block @ tm2
11	[PWR]	vdd*	2.700e+00		2.673e+00	2.727e+00	PASS	measure HV 'vddd' voltage value of 'u_s40srssa_coreana' block @ tm2
12	[PWR]	vdd*	2.700e+00		2.673e+00	2.727e+00	PASS	measure HV 'vddq' voltage value of 'u_s40srssa_coreana' block @ tm2
13	[PWR]	vdd*	2.700e+00		2.673e+00	2.727e+00	PASS	measure HV 'vddd_mon' voltage value of 'u_s40srssa_coreana' block @ tm2
14	[PWR]	vdd*	2.700e+00		2.673e+00	2.727e+00	PASS	measure HV 'vddio_mon' voltage value of 'u_s40srssa_coreana' block @ tm2

Total Criteria: 14

Figure 7. The third pdf report page - post simulation analysis

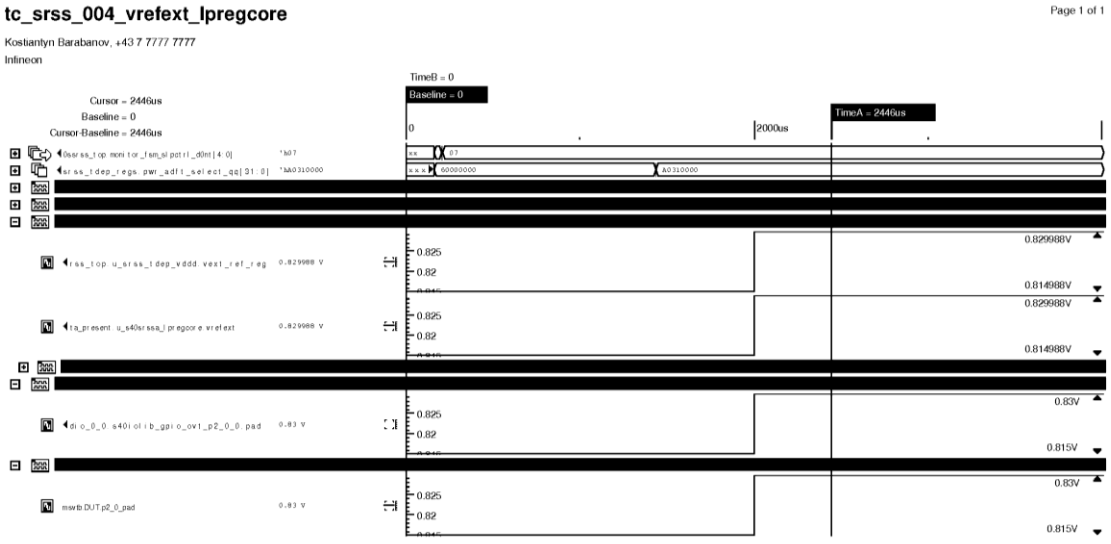


Figure 8. The fourth pdf report page – waveforms

REFERENCES

[1] M. S. Anwar, *PDF::Create - Create PDF files*, Version 1.46, MetaCPAN, Sep. 30, 2019. [Online]. Available: https://metacpan.org/pod/PDF::Create

[2] S. Simms, *PDF::API2 – Create, modify, and examine PDF files*, Version 2.047, MetaCPAN, May 18, 2024 [Online]. Available: https://metacpan.org/pod/PDF::API2

[3] SCHKN, *How To Search LDAP using ldapsearch (With Examples)*, DevConnected. [Online]. Available: https://devconnected.com/how-to-search-ldap-using-ldapsearch-examples/

[4] M. Fillion, *fpdf - A Python library for generating PDF documents*, Version 1.7.2, PyPI, Jan. 21, 2015 [Online]. Available: https://pypi.org/project/fpdf

[5] CodeRivers, "Working with FPDF in Python: A Comprehensive Guide," CodeRivers Blog, March 23, 2025 [Online]. Available: https://coderivers.org/blog/fpdf-python/

[6] TeX Stack Exchange, *What are good ways to convert EPS to PDF?*, Sep. 8, 2010. [Online]. Available: https://tex.stackexchange.com/questions/2872/what-are-good-ways-to-convert-eps-to-pdf

[7] H. Oberdiek, *The epstopdf Package Documentation*, Version 2.11, CTAN, Jan. 24, 2020. [Online]. Available: https://ctan.math.illinois.edu/macros/latex/contrib/epstopdf-pkg/epstopdf.pdf

[8] Hàn Thế´ Thành and others, *The pdfTEX user manual*, Version 1.40.27, CTAN, Feb. 12, 2025. [Online]. Available: https://at.ctan.niranjan.co/systems/doc/pdftex/manual/pdftex-a.pdf